Supporting Flexible Workflow Processes with a Progression Model

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By

Nicole Ann Stavness

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
University of Saskatchewan
Saskatoon, Saskatchewan S7N 5A9

Abstract

Users require flexibility when interacting with information systems to contend with changing business processes and to support diverse workflow. Model-based user interface design can accommodate flexible business processes by integrating workflow modelling with other modelling approaches. We present a workflow model, the progression model, to help in developing systems that support flexible business processes.

The progression model tracks a user's interaction with an application as a set of data elements we refer to as a *workflow transaction*. The steps a user takes to create a workflow transaction and the state of the workflow transaction at each step is made explicit. By making the workflow status and workflow transaction state explicit, the user can change the order of the steps in a process, manage multiple workflow transactions, keep track of data as it is accumulated, and so on. The intent is to provide the user with a mechanism to deal with partial information, interrupted and concurrent workflow transaction entry, and the processing of multiple workflow transactions.

This thesis describes the progression model, an XML-compliant notation to specify the progression model, and a prototype system.

Acknowledgements

I would like to acknowledge the remarkable support and guidance provided by my advisor, Dr. Kevin Schneider, throughout my graduate studies. The knowledge and experience that I have gained working with him will benefit my future with Computer Science. I would like to thank my thesis committee, Dr. Jean-Paul Tremblay, Dr. Gord McCalla, and Dr. Richard Long for their valuable feedback. The members of the Software Research Lab, in particular Jennifer Petrie, David Paquette, and Mark Watson, must also be thanked for many interesting discussions, positive support, and advice in general.

Finally, this achievement is a direct result of the love and support of my family. Thank you to my parents, Victoria and Lyle, for maintaining your faith in my abilities and consistently providing valuable encouragement. Thank you Ian for sharing your understanding and knowledge. Thank you Alyn for your positive outlook and listening ear. All of your support has been greatly appreciated.

For my family, Lyle, Victoria, Alyn, Ian, and Sara.

Contents

List of Figures

Chapter 1

Introduction

Business organizations perform work to achieve their business goals [24]. To consistently and productively ensure that the work progresses toward these goals, processes are formed that organize the work. These business processes can be implicitly performed or explicitly developed. However, they do require some guidance whether it be practiced work methods or automated systems to ensure consistency and accuracy. A business process is "a set of one or more linked activities that collectively realize a business goal, normally within the context of an organizational structure defining functional roles and relationships" [64]. Business processes are a required part of everyday business, whether explicit or implicit, as their fulfillment results in consistent and reliable work. Complex and data intensive business processes can be automated completely or in part by interactive software systems.

In the business environment, it is common for many changes to occur, such as government policy, products, services, competitors, customer demands, work methods, and so on. These changes in turn cause changes in the type of work and the way it is performed, which affects the business processes. This research focuses on how software can be created to support business processes throughout these changes. The types of flexibility that are being examined relate to the user's knowledge of the state of the process and the user's ability to manipulate the process.

A common term used to describe the orderliness and control of business processes

is workflow. The Canadian Oxford Dictionary defines workflow as "the organization of the sequence of industrial, administrative, etc. processes through which a piece of work passes from initiation to completion." The traditional branch of workflow research focuses on the workflow management system, which is "a type of interactive system that provides tools for coordinating work by managing the flow of information and responsibility within an organization" [23].

Our research focuses on the flow of work and workflow support for the individual user or groups of users within an information system. The type of information system that we have focused on intends to support the user by gathering the user's input and modeling that data as a workflow transaction. For example, the system could support a single user providing information to rent a car or multiple users filling out and approving a mortgage application.

Model-based user interface design is based on a description of application objects and operations at a level of abstraction higher than that of code [21]. It is a combination of many different aspects of the user interface that are explicitly formalized and represented in declarative models. It combines different user interface requirements in a conceptual representation of the interface. Some commonly included aspects of the user interface are: tasks, users, business objects, dialog, presentation, layout, and so on.

We used a model-based approach to utilize the many benefits of model-based user interface design. Model-based approaches make it easier to communicate with others as they provide a abstract description of the user interface. They facilitate systematic user interface implementation especially in automating some of the design

process. Additionally, the explicit qualities of these approaches allow for analysis.

1.1 Motivation

Business processes are modeled by information systems to assist users in achieving business goals. To contend with the changing aspects of business processes, these information systems require support for flexibility. Flexible workflow allows the user to keep track of data as it is accumulated, reorder the steps they take to complete the work, pass work on to another worker to be completed, view the completed work as well as the work that remains to be completed, interact with multiple business processes at one time, and receive feedback about the system constraints.

To support flexible workflow, a system requires support for workflow persistence, partial data, and tracking the workflow transaction. The particular data that we are interested in stems from the point of view of the user rather than the system designer or maintainer. This is so the user has a sense of the buildup of data occurring throughout the workflow. Therefore, our interest lies in the data that the user provides during their interaction with the system, including that directly entered as well as indirect data that is computer-generated in response to user input.

Passing work on to another worker for further completion also requires support for partial data and persistence. It is beneficial for the worker to receive the information that has already been accumulated, the actions that workers have taken throughout the workflow, and the requirements that each worker is assigned to fulfill. This allows each worker to view the work that has been completed, whom it was completed by, and what work still remains to be completed.

Reordering the workflow steps allows the user to perform work in a way that in part suits their individual circumstances. This requires support for partial data to ensure that upon completion of the work, all the necessary data is provided. Supporting partial data allows the user to bypass entering all the information that is requested at a step in the workflow process. The user may come back at a later time to provide the data or continue neglecting to provide the data. Support for partial data would include allowing for easy access to input missed or skipped data, as well as reminders to ensure that essential information is eventually provided. Therefore, the user is allowed to move throughout the system, entering data in the order that the user desires, and mapping out the workflow as they go.

Interacting with multiple business processes at one time allows the user to view multiple scenarios at one time. Also, they can increase consistency and efficiency by applying the same user actions to a batch of processes. This is a larger granularity of process manipulation that goes beyond the instance of one process.

Providing the user with feedback about the system constraints when the data of the workflow transaction is submitted to the application allows the user to have a more flexible interaction by not constraining their actions until the final submission. There may be some circumstances that require validating information from the application throughout the process, which can be accommodated. However, this still provides additional flexibility. Some workflow models attempt to support changes to the workflow process; however, they usually require the system administrator to execute the modifications [26]. They also tend to be fashioned for large workflow management systems that do not extend well to smaller information systems. Currently,

model-based user interface design research does not address workflow concepts.

1.2 Thesis Statement

By recording workflow transactions and supporting workflow actions, we can support flexible workflow interaction and manipulation in information systems. Flexible workflow allows the user to keep track of data as it is accumulated, reorder the steps they take to complete the work, pass work on to another worker to be completed, perform actions on multiple processes at one time, receive enriched error feedback, and view the completed work as well as the work that remains to be completed.

Observing the status of workflow and performing workflow actions can assist in achieving these flexible workflow goals. For the user to observe the status of workflow, they will need to see the recommended path of steps to complete the work; the state of completion of the step regarding whether it is inactive, incomplete, in progress, or completed; and the worker that is assigned to complete the work for each step. For the user to perform workflow actions there must be support for looking at the previous step, next step, saving work, recalling past work, and reordering the work steps.

1.3 Approach

Our approach is to design a model that makes the workflow information explicit. We have developed the progression model [60] that makes explicit the steps that a user performs and the data that is collected in a workflow transaction when using an information system. As a user progresses towards accomplishing a task or goal, the progression model infrastructure records each step, the state of the transaction, the

state of the workflow process, and the user's actions. To support the features of the progression model, we connect to the application with workflow transactions rather than with the traditional fine-grain event-based approach.

To specify the progression model, we developed an XML-based notation that is similar to a user interface description language. A user interface description language is "a high-level computer language for describing characteristics of interest in a user interface in relation to the rest of the interactive application" [59]. Several research groups have focused on developing such languages as UIML [1] and XIML [51]. Such languages have many benefits; they are easily conveyable to others through their explicit properties, are able to facilitate (semi-) generating code, have multi-platform operability, and so on [59]. Conferences such as Advanced Visual Interfaces 2004, that have devoted a workshop to examining and finding directions for UIDLs.

While incorporating increased flexibility to a system, the integrity of the software to provide the correct guidance in completing a business process must still be maintained. These issues have not been the focus of this research, but current research in standardized business process language [3] appears to provide some helpful insight.

A potential drawback to this approach may affect certain types of information systems. For business strategy-related reasons companies may not want to provide the user with additional flexibility. For example, in a flight ticket booking system, the user's personal information may be purposely requested in the first step to create user profiles. Therefore, the company would not want to have a software system that gives the user full control over entering partial data. Constraints would need to be developed that correspond to a company's business strategies.

1.4 Contributions

Through establishing the progression model, representing it in an XML-compliant notation, and implementing a system that demonstrates the functionality, we intend to support the following main contributions:

- Developing a model that makes workflow status and workflow transactions explicit; and,
- Expressing workflow concepts in an XML-based notation;

There are also some secondary contributions that are included in our model; however, they require additional research to be incorporated into the notation. The secondary contributions are as follows:

- Supporting workflow actions in information systems; and,
- Connecting the user interface to the application at the larger granularity of a workflow transaction.

The first contribution, developing a model, is described in detail in Chapter 3 - Progression Model. The second contribution is described in Chapter 4 - Progression Notation. Contribution 3, supporting workflow actions, includes support for traversing scenes, saving progressions, recalling progressions, and reordering scenes, and are explained in detail in Chapter 5 - Prototype and shown by example in Chapter 6. For the final contribution, we explored connecting to the application at a larger granularity level; namely, one transaction at a time, rather than for each event. This

facilitates workflow flexibility as the arbitrary event ordering constraints are virtually eliminated. However, the meaningful constraints that are required in the system can still be imposed. It only requires that a completed transaction be submitted to finalize the progression.

1.5 Simple Progression Example

To illustrate the concept of a progression, this section provides a simple example of one. A progression is composed of a transaction and a sequence of scenes. Each scene is composed of a user interface description and a workflow description. This is a simple progression of a user buying a fish from an online pet store. There are three scenes that describe a user selecting and adding a pet to their shopping cart: scene 1, Enter Website; scene 2, Select Fish; and scene 3, Add Fish to Cart. Our progression notation with associated screenshots is provided for each scene in Figure 1.1, Figure 1.2, and Figure 1.3, respectively. The example uses the Java J2EE Pet Store [55] example to illustrate the progression.

```
<progression>
    ...
    <transaction>
         <pet/>
         <name/>
         <amount/>
         <cost/>
         <total/>
    </transaction>
    <scene>
         <sceneMeta><sceneID> 1 </sceneID></sceneMeta>
         <userinterface>
              <link> Birds </link>
              <link> Cats </link>
              <link> Dogs </link>
              <link> Fish </link>
              ...
         </userinterface>
         <workflow>...</workflow>
    </scene>
    <scene><sceneMeta><sceneID> 2 </sceneID></sceneMeta>...</scene>
    <scene><sceneMeta><sceneID> 3 </sceneID></sceneMeta>...</scene>
    ...
<progression>
```

Figure 1.1: Buy a Fish Progression, Scene 1: Enter Website

```
<progression>
...
    <transaction>
         <pet> Fish </pet>
         <name> Large Angelfish </name>
         <amount/>
         <cost> $16.50 </cost>
         <total/>
    </transaction>
    <scene><sceneMeta><sceneID> 1 </sceneID></sceneMeta>...</scene>
    <scene>
         <sceneMeta><sceneID> 2 </sceneID></sceneMeta>
         <userinterface>
              <link> Birds </link>
              ...
              <link> Add to Cart </link>
         </userinterface>
         <workflow>...</workflow>
    </scene>
    <scene><sceneMeta><sceneID> 3 </sceneID></sceneMeta>...</scene>
    ...
<progression>
```

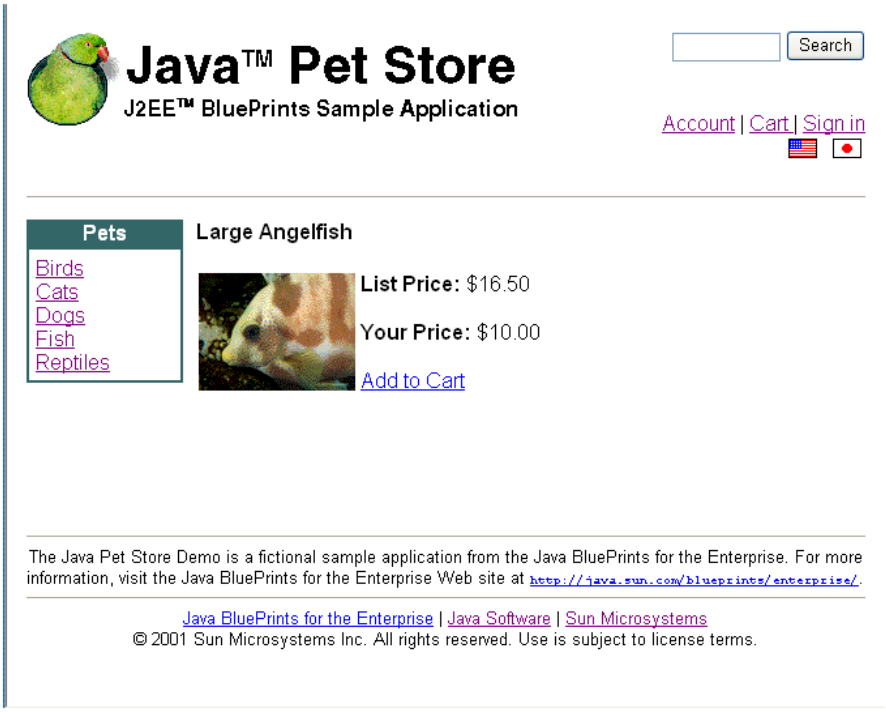

Figure 1.2: Buy a Fish Progression, Scene 2: Select Fish.

```
<progression>
...
    <transaction>
         <pet> Fish </pet>
         <name> Large Angelfish </name>
         <amount> 6 </amount>
         <cost> $16.50 </cost>
         <total> $99.00 </total>
    </transaction>
    <scene><sceneMeta><sceneID> 1 </sceneID></sceneMeta>...</scene>
    <scene><sceneMeta><sceneID> 2 </sceneID></sceneMeta>...</scene>
    <scene>
         <sceneMeta><sceneID> 3 </sceneMeta> </meta>
         <userinterface>
              <link> Birds </link>
              ...
              <button> Update Cart </button>
         </userinterface>
         <workflow>...</workflow>
    </scene>
    ...
<progression>
```

Figure 1.3: Buy a Fish Progression, Scene 3: Add Fish to Cart.

1.6 Outline

In this thesis, we present the progression model to capture a user's interaction with an information system by recording both the data the user enters and the steps the user takes to enter the data. The next chapter discusses related research that describes traditional workflow models; model-based user interface design and models that are commonly incorporated; and the benefits that user interface description languages can provide. The third chapter defines the progression model and identifies some benefits that can be gained. The fourth chapter outlines the features of the progression notation that is used to specify a progression model. The fifth chapter explains the progression analyzer prototype system, that is used to display information about a progression. The sixth chapter depicts an example to demonstrate the progression analyzer. The final chapter concludes with a discussion of the drawbacks of this approach and the future direction of research.

Chapter 2

Related Work

The previous chapter introduced the motivation, thesis statement, approach and contribution of our research. In this chapter we will describe current workflow research. We also look at model-based approaches for specifying interactive systems, and in particular task model approaches. We will examine user interface description languages and the Business Process Execution Language to specify user interface elements and business processes. Then in the next chapter we will define the elements that make up the progression model.

A number of approaches have been used to model business processes. Workflow models are used to describe the flow of work in an organization and with external organizations. Task models define the possible actions available for a user to accomplish a goal. Use cases describe stories that define ways a user can use a system. Software evolution can assist in incorporating additional flexibility into current systems. Model-based user interface design aids in creating interactive software that considers multiple factors, such as users, tasks, and so on. User Interface Description Languages help define user interfaces linguistically with a general trend to do so in an XML-compliant way. The Business Process Execution Language describes a notation for automating business processes in web services applications [17]. Each of these approaches is discussed in this section.

2.1 Workflow

The central concept of workflow modeling is appropriately based on work. Four dimensions for describing the flow of work are action structure, actors, tools and information [62]. Actions are the steps required to make the necessary changes to accomplish a goal. Actors, defined by roles, perform work by carrying out their choice of actions. Tools are the resources that are used by the actors to complete the actions. At the required abstraction for workflow, tools are usually applications or components within the systems. The information that is acted on is represented by objects. Objects are also used for modeling useful contextual information used in actions.

The Workflow Management Coalition (WFMC) defines workflow as "the automation of a business process, in whole or part, during which documents, information, or tasks are passed from one participant to another for action, according to a set of procedural rules" [64]. The details of a specific business process are defined in a process definition. This includes the sequences of activities and associated relationships; start and finish criteria; and information such as executers of manual or automated activities, procedural rules, and control data. The process definition may also include sub-process descriptions.

The entire process can be managed by a workflow management system that controls the automated functionality [64, 8, 53, 65, 22]. The workflow management system defines, creates, and manages workflow execution through the software. Workflow engines, for interpreting the process definition, interact with the human or

automated participants. Therefore, workflow can either have most of the procedural rules defined before the process begins, as in conventional workflow management systems, or the procedural rules can be changed or created during the process operation. The latter is also referred to as Adaptive Workflow Management [41].

2.1.1 Dynamic Workflow

There are many variations on how to support the dynamic aspect in workflow processes. Dynamic process support is a current research area as there is demand for the workflow management systems to keep changing due to support changes to workflow during the enactment of the process, especially as some of the larger workflow systems manage timely workflow processes. The following research projects exemplify some of the different approaches.

Bernstein [10] views procedure-like, routine processes that are statically supported on one end of a continuum, with highly unspecified, dynamic processes on the other end. A tool is presented to bridge the gaps between these extremities to contend with processes that fall at any point along this continuum. The example scenario that Bernstein outlines begins with an account manager, Heidi, of a computer company needing to send out a new server for a Swiss company within 48 hours. Heidi's traditional workflow system cannot accommodate the task as the truckers are on strike. She must use email, telephone, or fax to accomplish the task, which puts the burden of contextual sense-making on whoever Heidi contacts. According to the specificity continuum, process specificity changes over time, much like Heidi's situation in this scenario. Her task begins with having a reasonably well-specified

solution process. However, when it is discovered that the truckers are on strike, the known solution is no longer applicable. Therefore, Bernstein claims that a support system that allows the processes to change from well-defined to more flexible can be beneficial in many scenarios as with Heidi's situation.

Haake and Wang [24] propose a cooperative hypermedia system, with process support through a meta-model, which integrates the efforts towards communication, coordination, and cooperation in workflow systems. They identify that some systems help support groups of users in the coordination of their collaboration, that is the planning and scheduling of business processes, while others support the carrying out of the actual work, such as the cooperative execution of the process activities. However, there are no systems that integrate both of these important aspects. They attempt to achieve this through a hypermedia-based process support system. They support a variety of process structures, for instance in an unstructured brainstorming session, they create a white board space. Whereas for a large mandatory scheduled process like preparing a project deliverable, a process space is created in addition to the task space of the document in preparation, to guide the task flows.

Heinl et al [26] describes a two-part classification that defines types of possible flexibilities that may be desired in workflow management applications. *Flexibility by selection* provides the user some leniency in executing a process by offering multiple execution paths. Alternatively, *flexibility by adaption* provides the ability to add extra execution paths through additional functionality and tools that allow the workflow type to change and integrate during runtime. However, this framework requires modelers to create and add workflow types and execution paths.

Mangan and Sadiq [35] look at executing processes with a partially defined model where the full and often unique specification is made at runtime. They construct the partial model with a set of fragments, either tasks or sub-processes; modeling constructs such as sequence or multiple executions; and a set of constraints to define the rules that govern how valid instances can be made. Three types of constraints are defined: selection, terminations, and build. The example that they depict in the paper consists of the process of choosing elective courses in a tertiary institution. The fragments are the courses. Selection constraints dictate which fragments are available for building at a particular time. For instance, a course is not available to be taken in the same semester as its prerequisite course. Termination constraints define the end of the process goal in the absence of a termination task. For example, once the total number of courses is met, the course selection process is complete. Build constraints specify additional constraints that affect the building of the process instances. For instance, some honours courses may require a minimum grade average or some courses may only be available in alternate semesters. Therefore, the modeler can put together the fragments according to the constraints.

Some workflow concepts that are common in many systems have been identified by Manolescu [37]. This research has focused on applying workflow to object-oriented systems. The following concepts are identified as important to workflow:

- *Monitoring* for contributing information about the circumstances of workflow during execution;

- *History* of workflow actions for evaluation or recovery;

- *Persistence* to save the historic information and provide access to it;
- *Manual Intervention* for changing the order that activities are performed in as they are performed;
- *Worklist* to coordinate the activities among the workers; and,
- *Federated Workflow* to address the issue of how workflow systems interoperate.

2.1.2 Discussion

Although these approaches facilitate dynamic intervention in workflow model enactment, they usually require intervention from the system administration or model designer. Unless the end user has in-depth knowledge of the system, they cannot make any changes to the process themselves. This does not allow quick and easy changes to the business processes. Therefore, these systems are rarely utilized in practice on a broad range. Many of the workflow management systems in use are created for large organizations on a one-time basis. The architectures tend to be narrowly focused to accommodate the particular application for which they are tailored [36]. The types of organizations that would deploy these systems also tend to have the manpower to support the process changes from the system administrator. They do not seem feasible for smaller companies or smaller systems that would greatly benefit from having the end user make the process changes.

2.2 Task Models

Task Models are logical descriptions of activities that are designed to be carried out in reaching users' goals in interactive systems [47, 31]. Task models are useful in

most phases of software development. More specifically, they are helpful for understanding an application domain, recording the results of interdisciplinary discussions, designing new applications consistent with the users' conceptual model, analyzing and evaluating the usability of an interactive system, supporting the user during a session, and documenting interactive software.

There are three types of task models that are useful in system development [44]. System task models describe how activities should be performed in a current system, such as in understanding a system or performing a usability evaluation. Envisioned task models determine how users should interact with a new system at a coarser grained level. This is often used for considering new design solutions. Finally, user task models show how users think that tasks should be performed in order to reach their goals.

2.2.1 Task Model Approaches

Each approach includes tasks, which are the activities that must be performed to reach a goal [44]. A goal is the desired change in the state of a system or an attempt to retrieve information from a system. Each task is associated with only one goal, where the goal is accomplished by completing the task. Beyond these basic concepts, there are many different approaches to task modeling such as Hierarchical Task Analysis [4], GOMS [14], UAN [25], and ConcurTaskTrees [48].

Hierarchical Task Analysis

Hierarchical Task Analysis (HTA) is based on describing the set of goals, tasks, and operations in logical structures of different levels [4]. HTA tries to capture the steps

that are undertaken by the users when interacting with the system. The goal is the concept that the user wants to attain related to the state of the system. Tasks describe the manner in which a goal is achieved, and operations are the lowest level units of behaviour performed within a task. Thus, a goal that is attained by a person has a task that dictates how the goal is achieved and operation(s) that indicate what is actually done within the constraints of the task. Goals are described at various lower-levels of abstraction, which are referred to as the sub-goals.

GOMS

GOMS (goals, operators, methods, selection) depicts procedural knowledge or how-to-do-it knowledge [14, 29, 30]. GOMS is based on a cognitive model, the human processor. The model is made up of a set of underlying memories, processors, and principles that dictate behaviour. GOMS uses a hierarchical description of operators that are performed to reach a goal. Operators are the base elements that cannot be broken down further, which can extend to various levels of abstraction depending on the needs of the system. They represent the basic perceptual, motor, and cognitive acts. Methods refer to a sequence of goals and operators that lead to a high-level goal. Furthermore, selection rules dictate whether one method should be used over another.

User Action Notation

UAN (User Action Notation) also follows a hierarchical structure [25, 57, 58]. It intends to communicate design by allowing designers to describe the dynamic behaviour of graphical user interfaces. The tasks are represented asynchronously with operators that denote the temporal relationships between them. Another part of

UAN associates each atomic task with a table that indicates user actions, the system feedback, and the state modifications requested to perform it. Unfortunately, the tasks must follow a left–to–right ordering to interpret the tables, which can be rigid and insufficient.

ConcurTaskTrees

The ConcurTaskTrees notation was created to support engineering approaches to task modeling [48, 39, 45]. Temporal relationships are also incorporated for enabling, concurrency, disabling, interruption, and optionality. Additional information can also be associated with each task such as type, performance allocation, and required objects to be manipulated. The notation also provides the ability to allow designers to describe concurrent tasks beyond the sequential capabilities of GOMS or HTA. Additionally, synchronized tasks where the output information of one task is the input information of another are supported beyond UAN's ability. Therefore, ConcurTaskTrees is set apart from others through this ability to specify complex behaviours.

ConcurTaskTrees is based on the LOTOS formal notation, which allows users to describe event-driven behaviours and state modifications. The ConcurTaskTrees extension includes new operators to express dynamic behaviours. It is also meant to be easier to use than the text-based LOTOS.

2.2.2 Discussion

There are many different types of task models that cater to different levels of granularity of task descriptions. In relation to business processes, task models describe

the paths of activities available to reach the user's goals. Systems are usually built from system designer envisioned task models. However, business process support would benefit from the user's own task model, where a user indicates how they think the task should be done. The designer could attempt to predict all the possible task paths that the user may want to take. However, even if the designer was able to accurately foresee all possible task paths, the resulting task specification would be extremely large. These specifications provide more detail than is necessary for the designer, which can impede the helpfulness of this approach.

2.3 Use Case Analysis

Use cases are thought of as the ways in which a user uses a system. Use cases are used to organize the requirements of the system from the user's perspective. Although the specifics of putting a use case together may vary, this broader definition manages to endure. There are differences of opinion regarding the formalism, structure, and consistency of uses cases. The main purpose of use cases is to build system requirements [15, 16]. The contents should follow a consistent style in a semi-formal structure with multiple scenarios per use case. Therefore, a use case is a collection of possible sequences of interactions between the system under discussion and its external actors, related to a particular goal.

Actors may be human users or computer systems. A primary actor is one that requires help from the system to achieve its goal [15, 16]. A secondary actor, on the other hand, provides the help that a system requires to satisfy its goal. Actors can be either internal or external to the system. External actors can be one person,

multiple people or another system. Conversely, an internal actor refer to the system in design, a subsystem or an object. Each actor has responsibilities that are carried out through the set goals, which are accomplished by actions. An action is the triggering of an interaction with another actor, whether human or computer, calling upon one of the responsibilities of the other actor.

Furthermore, an action is either the sending of a message or a sequence of actions [15, 16]. A sequence has no branches or alternatives. It is just the collection of actions used to describe the past or future depending on particular conditions. The sequence of actions is also referred to as a *scenario*. When describing a system, scenarios are gathered that occur during one action. Accordingly, a collection of scenarios constitutes a use case.

It can be difficult to determine the scope of a scenario or use case. One way of scoping is to ensure that all the actions relate to the same goal. Another is to begin by triggering actions toward a goal and end when the goal is accomplished or abandoned, fulfilling the system's responsibilities. Use cases not only handle situations when a goal is achieved, but also when a goal fails.

2.3.1 Use Case Maps

To bridge the gap between use cases and more detailed design views, Use Case Maps are identified for depicting scenarios [2]. They facilitate reasoning about the early stages of design related to distributing functions to components [50]. This visual notation, which is shown in Figure 2.1 assists in describing causal relationships between responsibilities of system components. This figure shows a use case map

where a web server accesses a database [54]. Scenario paths are shown as lines joining responsibilities in a casual sequence running over components, connecting to stubs in the diagram. Stubs represent containers for sub-maps called plug-ins. The paths execute responsibilities within components along the way, depicting the functionality associated with the execution of the scenario.

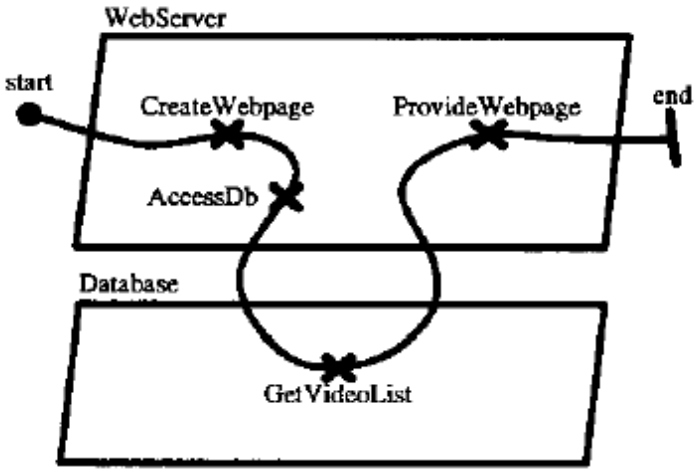

Figure 2.1: A Use Case Map of a webserver accessing a database [54]

An aspect particularly beneficial for business processes is the ability to combine and integrate scenarios to model dynamic runtime behaviour [11]. The stubs can be static, containing one plug-in map, or dynamic, containing multiple plug-in maps. The selection of a plug-in map in the dynamic stubs can be determined at runtime according to selection policies. The selection policies are usually defined in a formal language. Although the various paths are statically defined, the selection of the path is indicated dynamically.

Use case maps do represent dynamic use case scenarios. However, the formal notation, which is further explained in [11], requires fine-grained details and large

specifications.

2.4 Business Process Execution Language

The Business Process Execution Language (BEPL) was created to describe business interactions consisting of sequences of peer to peer message exchanges, both synchronous and asynchronous, within stateful, long running interactions involving two or more parties [3]. BPEL is a language that defines a notation for specifying business process behaviour based on web services. Web services are self-contained, self-describing modular applications that can be published, located, and invoked across the web[5, 19, 6]. Since they are often used to pass business logic amongst companies via the web, BPEL was created to specify the logic. BPEL provides constructs for sequences, loops, conditionals, fault handling, and concurrent execution of operations.

An example of BEPL follows. The `partner` elements are the those that interact with the business process. The `faultHandlers` element indicates ways to handle and recover from errors. The `compensationHandler` allows the designer to implement actions for irreversible errors. The `activities` are the actions being carried out within a business process. The `correlationSets` element determines which instance an incoming message is directed.

```
<process name="ncname" targetNamespace="uri"
    queryLanguage="anyURI"?
    expressionLanguage="anyURI"?
    suppressJoinFailure="yes|no"?
    enableInstanceCompensation="yes|no"?
    abstractProcess="yes|no"?
    xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/">
```

```
    <partnerLinks>?
    <!-- Note: At least one role must be specified. -->
         <partnerLink name="ncname" partnerLinkType="qname"
         myRole="ncname"? partnerRole="ncname"?>+</partnerLink>
    </partnerLinks>
    <partners>?
         <partner name="ncname">+<partnerLink name="ncname"/>+</partner>
    </partners>

    <variables>?
         <variable name="ncname" messageType="qname"? type="qname"?
         element="qname"?/>+
    </variables>

    <correlationSets>?
         <correlationSet name="ncname" properties="qname-list"/>+
    </correlationSets>

    <faultHandlers>?
    <!-- Note: There must be at least one fault handler or default. -->
         <catch faultName="qname"? faultVariable="ncname"?>*
         activity  </catch>
         <catchAll>?  activity   </catchAll>
    </faultHandlers>

    <compensationHandler>?   activity   </compensationHandler>

    <eventHandlers>?
    <!-- Note: There must be at least one onMessage or onAlarm handler. -->
         <onMessage partnerLink="ncname" portType="qname"
         operation="ncname" variable="ncname"?>
              <correlations>?   <correlation set="ncname"
              initiate="yes|no"?>+   <correlations>  activity
         </onMessage>

         <onAlarm for="duration-expr"? until="deadline-expr"?>*
         activity  </onAlarm>
    </eventHandlers>
    activity
</process>
```

2.5 Model-based User Interface Design

Model-based user interface design [20, 13, 43, 28, 38, 56, 42, 63, 18] is intended to assist in designing user interfaces with a more formal, computer supported methodology rather than the more common information paper design, such as storyboarding or pictive [21]. It describes the application model as an executable specification, yet at an abstraction level higher than that of code. It attempts to explicitly represent knowledge that is often hidden in the application code.

To describe the application, many different models are incorporated in model-based user interface design. For instance, Puerta and Eisenstein [18] describe an interface model, which is an ordered collection of all the relevant elements of the user interface. There elements are grouped into models including task model, domain model, user model, presentation model, and dialog model. A task model describes the activities that the user can perform through the application's user interface. A domain model defines the objects of the application that the user can view, access, and manipulate through the user interface. A user model identifies the types of users of the application by the user's attributes and roles. A presentation model represents the visual, haptic, and auditory elements that are offered to the user through the user interface. A dialog model indicates how the presentation model interacts with the user.

Although many aspects of the user interface are considered through these models, there are circumstances that require flexibility in the user interface, such as switching devices or supporting business processes. When referring to flexibility in the user

interface, the term plasticity or plastic user interfaces is used.

Calvary et al [12] revises the reference framework that is a user interface design model intended to create plastic user interfaces. Within the context of Human Computer Interaction, plastic is defined as the capacity of an interactive system to withstand variations of context of use while preserving usability. Therefore a plastic user interface should be able to adapt to context changes while preserving usability. The reference framework attempts to incorporate many different models, as shown on the far right in Figure 2.2. The concepts model describes the domain, similar to domain modeling. The task model outlines how the user can perform a task to achieve the task goal. The platform model describes attributes of the device in use, such as computer screen size. The environment model indicates influences from the external environment, such as noise level in the room. The interactors model describes the "resource sensitive multimodal widgets" available for producing the concrete user interface. The evolution model specifies static change for a context and the conditions for entering and leaving a context. Although many interaction aspects are included in this framework, workflow concepts have not been incorporated.

The following are some key benefits that have been identified of model-based user interface design [21]. They can help support multiple interfaces by generating different layouts from the same model. They support separation of concerns by modularizing the interface model from the application core. They assist in analyzing the system for consistency and completeness through the mathematical aspect of the specification language. They help in describing input scenarios and testing the user interface's usability. They also aid in designing the user interface with a user-centered

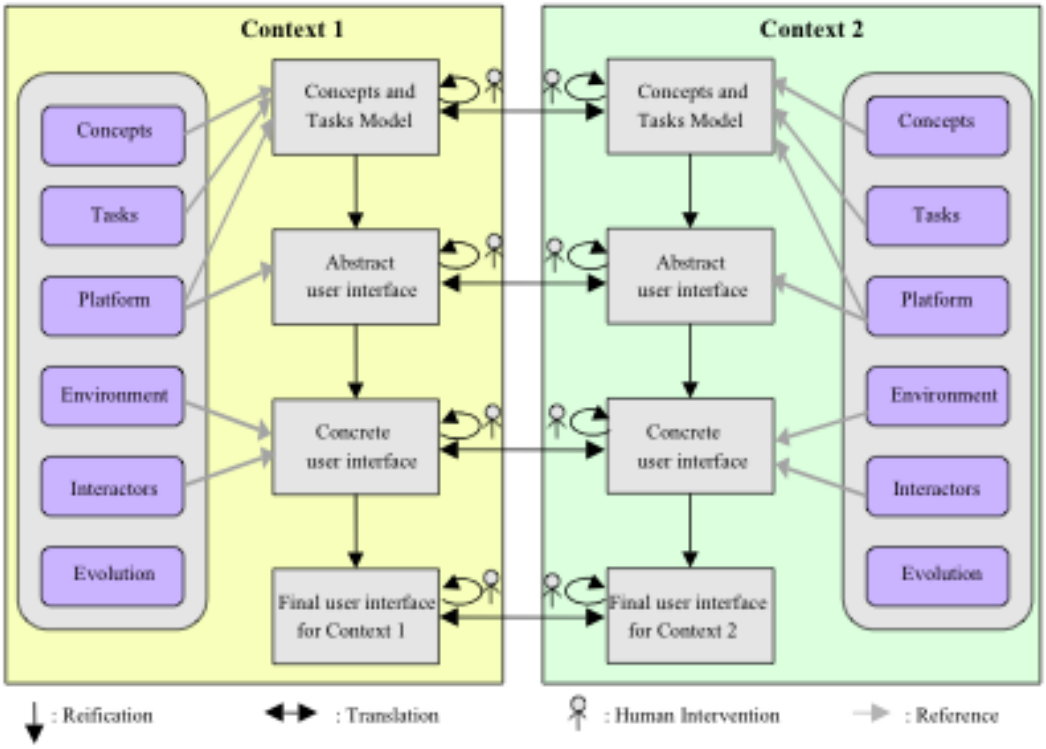

Figure 2.2: A reference framework for model-based user interface design [12].

focus. However, these models are not often utilized in practice. They tend to use complex languages to specify the model, which are timely and complex to implement. Also, the support tools have difficulty supporting the connection between the user interface and the model. Some automatically generate the user interface from the model, but do not allow the designer to adjust the generated user interface. Others that do allow user interface modifications do not update the model with the changes. As well, they often require the user to create the model before they can start on the visual interface, which can restrict the designer's abilities.

2.6 User Interface Description Languages

A user interface description language (UIDL) [40, 27] is intended to capture the details of what a user interface could or should consist [59]. UIDLs describe characteristics of interest of a user interface according to the rest of an interactive system with a high-level language. A UIDL allows the user interface to be specified independently from any target language that would be used to implement the user interface. Many UIDLs have been conceived that contain different features and focus on different levels of granularity. Souchon and Vanderdonckt [59] have identified and analyzed a number of XML-compliant languages for defining user interfaces including UIML [1], AUIML [9], XIML [51], Seescoa XML [33, 32], Teresa XML [49], and WSXL [7].

User Interface Markup Language (UIML) allows the user to specify the user interface in general terms and then render it according to a style description [1]. It is an XML-compliant language designed to create user interfaces for any device, any target language, and any operating system. It is also designed to be independent of any user interface metaphor such as graphical user interfaces or voice-response. The user interface designer must design an interface for each device and represent it in UIML. However, UIML has approximately thirty tags, which is much less than many other UIDLs and also easier to quickly specify user interfaces. This language allows the designer to specify the appearance, user interaction, and application connection of the user interface. Figure 2.3 which shows the logical structure of an interface description in UIML, follows this skeleton [1].

```
<?xml version="1.0" standalone="no"?>
    <uiml version="2.0">

    <interface name="Figure5" class="MyApps">
         <description>...</description>
         <structure>...</structure>
         <data>...</data>
         <style>...</style>
         <events>...</events>
    </interface>

    <logic>
    </logic>
</uiml>
```

Figure 2.3: The logical structure of an interface description in UIML [1].

Abstract User Interface Markup Language (AUIML) focuses on describing the desired user interaction in terms of its purpose rather than appearance [9]. It is intended to allow designers to focus on the semantics of the interactions rather than the particular devices that need to be supported. This language describes the user interface in terms of manipulated elements, interaction elements, and actions. Manipulated elements are a data model of the information required for a particular interaction. Interaction elements are the presentation models that indicate the look of the user interface. Actions describe the dialogue of events between the interface and data. Additionally, the designer can decide the degree of specificity of the renderer, whether to define the display explicitly or only an interaction style where the renderer decides the layout.

The eXtensible Interface Markup Language (XIML) affords the ability to describe a user interface without concern for the implementation [51]. XIML intends to sup-

port abstract and concrete functionality across user interface design, development, operation, management, organization, and evaluation. There is support for mapping from abstract to concrete aspects. XIML defines five basic interface elements: task components, domain components, user components, dialog components, and presentation components, but is not limited to these elements. A talks component captures the process or user tasks of the user interface. A domain component defines all the objects and classes. A user component looks at the characteristics of the users that use the application. A dialog component defines the user interaction with the user interface. A presentation component indicates the appearance and layout of the user interface. Additionally, XIML includes attributes and relations to connect with the elements. Figure 2.4 shows the basic representational structure of the XIML language [52].

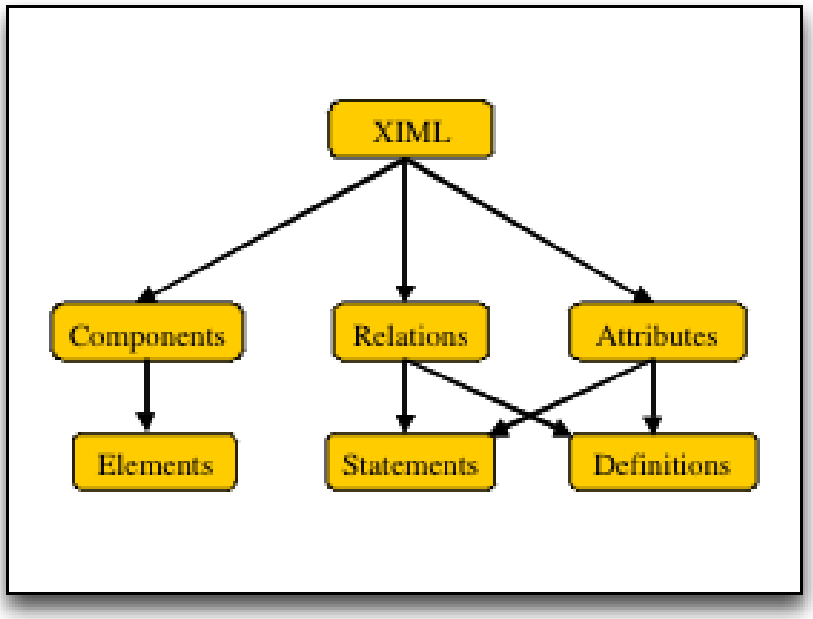

Figure 2.4: The basic representational structure of the XIML language [52].

Software Engineering for Embedded Systems using a Component Oriented Approach (Seescoa XML) defines an XML description to express an abstraction of the user interface using Java User Interface components [33]. Seescoa XML has an XML description that outlines an abstraction of the user interface as a hierarchy of Abstract Interaction Objects (AIO). The platform independent AIO is mapped onto a platform specific CIO (Concrete Interaction Object). There is an XSLT for each system that maps the AIO to the CIO based on the constraints of each platform. A downfall of this language is that it only creates Java user interfaces. Figure 2.5 shows two sections of Seescoa XML [34]. The first is a user interface description of a single camera component and the second is a user interface description of a mosaic component. The separate components are combined to create the entire user interface.

Teresa XML provides a facility to support the design and generate a concrete user interface for a specific type of platform [49]. The Teresa XML language has an XML-description of the CTT notation (task model description) and a language for describing the user interface. It specifies how the AIOs composing the user interface are organized as well as the user interface dialog. In addition to the AIO, which describes the static organization of the user interface, there are also zero or more connections that dictate the relationships between the presentation elements of the user interface. TERESA is the tool that utilizes Teresa XML to generate task models, abstract user interfaces and running user interfaces. Figure 2.6 shows an example of Teresa XML for describing an XML version of the ConcurTaskTrees notation [49].

```
UI description of a single camera component:
<group name="camera2">
    <interactor>
         <videowidget name="video">...</videowidget>
    </interactor>
    <interactor>
         <range name="zoomrange"><action>
              <func service="Surveillance.Controls">setFocus</func>
              <param name="camera2"/><param name="zoomrange"/>
         </action></range>
    </interactor>
    <interactor><range name="focusrange">...</range></interactor>
    <interactor>
         <button name="camera1_onoff"><action>
              <func service="Surveillance.Controls">switch</func>
              <param name="camera2"/><param name="camera1_onoff"/>
         </action></button>
    </interactor>
</group>

UI description of a Mosaic component:
<ui>
    <title>Camera mosaic</title>
    <group name="mosaic">
         <group name="camera1">&CAMERA1</group>
         <group name="camera2">&CAMERA2</group>
         <group name="camera3">&CAMERA3</group>
         <group name="camera4">&CAMERA4</group>
    </group>
</ui>
```

Figure 2.5: A partial example of a Seescoa XML specification. The first section describes a camera component. The second section is the mosaic component that groups individual components together [34].

```
<!ELEMENT Task  (Name, Type, Description, Platform*, Precondition?,
TemporalOperator?, TimePerformance, Parent?, SiblingLeft?,
SiblingRight?, Object*, SubTask*) >

<!ELEMENT Name  (#PCDATA) >
<!ELEMENT Type  (#PCDATA) >
<!ELEMENT  Description (#PCDATA) >
<!ELEMENT  Platform (#PCDATA) >
<!ELEMENT  TemporalOperator (#PCDATA) >
<!ELEMENT  TimePerformance (#PCDATA) >
<!ELEMENT Max (#PCDATA)  >
<!ELEMENT Min (#PCDATA)  >
<!ELEMENT  Average (#PCDATA) >
<!ELEMENT  Parent EMPTY >
<!ATTLIST  Parent name CDATA #REQUIRED>
<!ELEMENT  SiblingLeft EMPTY >
<!ATTLIST  SiblingLeft name CDATA #REQUIRED >
<!ELEMENT  SiblingRight EMPTY >
<!ATTLIST  SiblingRight name CDATA #REQUIRED >
<!ELEMENT  SubTask (Task*) >
```

Figure 2.6: The Teresa XML for describing the ConcurTaskTrees notation. [46].

Web Services Experience Language (WSXL) focuses on a web services model to interact with web applications [7]. WSXL attempts to build web applications for a wide variety of channels as well as create web applications for others. It enables applications to be built out of separate presentation data and control components. This helps put together alternative versions of the components to meet the requirements of separate channels, users, and tasks.

The User Interface Description Languages referenced above do not address workflow issues.

2.7 Discussion

Although task models, use cases, and UIDLs were not specifically designed to ensure flexibility in automating business processes, they provide insight into designing and changing software. Workflow models were designed for supporting business processes; however, they are only used in large, multi-user systems. Therefore, when examining flexibility of business processes in information systems, we need to take a slightly different approach.

Due to the similarities in process modeling, there are often discrepancies between the foundational difference between workflow models and task models aside from the types of systems they attempt to model. As noted earlier, Traetteberg [62] identifies workflow models as useful for group or organization interaction, while task models focus on individual users. Our examination of these models, and the systems they are used for, has resulted in a more substantial differentiation. Workflow systems inherently focus on the management of task accomplishment processes. Many of the components that are commonly included in workflow models are not found in task models. According to Manolescu [37], these components include process monitor, history, persistence, manual intervention, work list, and federated workflow. These components go beyond the actual process and focus on how the process is completed and how to examine it thereafter.

In this chapter we have summarized the current research pertaining to workflow, task models, use cases, the Business Process Execution Language, model-based user interface design, and user interface description languages. We have seen that

task models and use cases assist in designing software; however they do not address workflow issues. Model-based user interface design looks at user centric software design, but also does not support a workflow focus. User interface description languages help describe the user interface, but do not define workflow concepts. The Business Process Execution Language can be helpful in specifying business process constraints, but they do not make the workflow steps and workflow transaction explicit. In the next chapter, we introduce our model, the progression model, that incorporates workflow concepts to add flexibility.

Chapter 3

Progression Model

The previous chapter discussed the current research in workflow as well as model-based approaches for designing software systems. User interface description languages were also examined for specifying user interfaces. In this chapter, we will present the progression model. Next, we will identify some of the benefits that can be gained from utilizing the progression model including information orientation, immediate updates, historical review, concurrent multiple progression comparison, progression batches, and scene reordering. In the next chapter, we will define the progression specification language used to specify a progression model.

The progression model [60] incorporates some of the managing concepts of workflow to increase the flexibility in information systems. The progression model makes explicit the steps and transactions a user undertakes when using an information system. As the user progresses towards accomplishing a task or goal, the progression model infrastructure records each step and the state of the transaction and workflow.

Making the steps and transactions explicit allows the user to group transactions into batches for later processing, store partial transactions for later editing, and browse historical progressions. Linking the steps in the workflow directly to the transaction provides a means to integrate the process model and the data model in one coherent model. This enables the support of the flow of work for an individual user by supporting new interactions. Consequently, new interactions can provide

flexible business process support.

In this section we will introduce the basic concepts and terminology of the progression model. Then we will discuss some of the benefits of this approach.

3.1 Terminology

The following definitions describe the key elements of the progression model and how they relate to each other.

Progression. A *progression* is a workflow transaction and a sequence of *scenes* in a process to create a workflow transaction, that is

$$progression = < workflow\_transaction, scenes >,$$

where $scenes = [\ scene_1, \ldots, scene_n\ ]$

Scene. A $scene_i$ corresponds to a step in a progression. A scene captures the process and associated data as a user performs actions throughout a progression. Each scene of a progression is associated with the user interface, $user\_interface_i$, the current state of the workflow status, $workflow\_status_i$, and the workflow constraints, $workflow\_constraints_i$, therefore,

$$scene_i = < user\_interface_i, workflow\_status_i, workflow\_constraints_i >$$

Workflow Transaction. The *workflow transaction* models the accumulation of data at each point in the progression. Each transaction is made up of a series of data elements, *element*, that are accumulated throughout the progression by user actions, that is

$$workflow\_transaction = [\ element_1, \ldots, element_n\ ]$$

As the scenes are enacted, the data element additions, deletions, and changes are

reflected in the transaction. For instance, if the user is filling out a wizard form, at every submission the new information is added to the transaction.

The *workflow_transaction* also models the status of the elements and the communication feedback from the application. As a user makes inputs, the *workflow_transaction* shows the status of completion. For example, a text field *user_interface* component status may begin as "Inactive", then when the user enters text it changes to "In Progress", and when the user exits the text field, if there was text entered, the status changes to "Complete".

The *workflow_transaction* can be validated with the application at any time. This determines if the user input is the correct type, whether it exists in the database where applicable, and so on. In addition to validation, feedback on timing and effects of change in other *workflow_transaction* elements are supported. For example, when booking a concert ticket, after selecting the number of tickets, the user only has a certain amount of time to fill out the billing information before the tickets are unreserved. If the time runs out, the status of the ticket elements becomes timed-out. Another example of status change feedback is when a change in the value of one element causes the status of another element to change. For example, if a customer edits the number of airline tickets booked in the workflow transaction after approval is given for the tickets, the approval from the airline becomes stale until the changes are approved by the airline.

User Interface. The $user\_interface_i$ is a rendering of the user interface components for the i^{th} scene. The user can perform user actions according to the user interface components, such as a text field or select box, to manipulate and complete

the progression.

Workflow Status. The *workflow status* is the state of workflow for a scene. The status includes information such as the recommended scene path, worker assignment, status of the scene completion. The recommended scene path outlines the initial scene order, which can be modified with workflow actions. The worker assignment designates which worker is assigned to complete each scene. The status of scene completion shows the states of each scene, such as "Inactive", "In Progress", "Incomplete", "Complete".

Workflow Constraints. The *workflow constraints* specify the restrictions that are placed on the progression by the application designer to meet the business requirements of the company. Some examples include scene ordering restrictions, user permissions, scene path merging or splitting, other scene path manipulations. An example of ordering restrictions is when the user is required to enter the air flight information before the ticket booking is processed and confirmed. Therefore, the confirmation scene could not be reordered before the information entry scenes. An example of user permissions is when the teller user in a banking system is not permitted to rearrange the fund approval scene order. Only the manager has permission to make this type of change to ensure quality control according to the bank's business goals. Other types of scene path manipulation coincide with process model splits, joins, concurrency.

User Actions. The user can perform actions to the user interface, workflow transaction, workflow status, and workflow constraints. There are three types of user actions: user interface action, transaction action, and workflow action. Each action

changes the values or state of a user interface component, which results in the change to the user_interface, workflow_transaction, workflow_status, or workflow_constraints.

User Interface Action. A *User Interface Action* occurs when the user performs an action to a user interface component to add, remove, or modify input, such as a filling in a text field or choosing from a select box. *User actions* can result in changes that update the workflow transaction with data from the user interface, change the workflow transaction item status when data is entered in the user interface, or change the workflow status as a scene is being completed through the user interface.

Transaction Action. A *Transaction Action* occurs when the user performs an action to the transaction to change the element values. Transaction actions can result in change that updates the user interface with data entered in workflow transactions, changes workflow transaction item status when data is entered in the workflow transaction, or changes the workflow status as a scene is being completed through the workflow transaction.

Workflow Action. A *Workflow Action* occurs when the user performs an action to the workflow status or constraints, which updates or manipulates the progression workflow. Workflow actions can result in changes such as navigating to another scene regardless of the recommended scene order, reordering scenes to change the recommended scene order, and changing the worker assigned to complete the scene.

3.2 Benefits of Recording Progressions

An information system is developed to support an organization's business processes. This requires a high degree of flexibility, which has been traditionally difficult to

support. The process and data information that is captured through the progression model can be used to support flexible business processes. It is facilitated by displaying the transaction to the user, in addition to the original interface, accompanied by new functionality. Through opening the model to the user in this way, a number of new interactions become available to the user. Some types of use interactions that can be enabled are information orientation, immediate updates, historical review, concurrent multiple progression comparison, progression batching, and scene reordering.

3.2.1 Information Orientation

By visually observing the transaction, the user is able to see the information being built up while the progression is enacted. This provides a reference for the user to ensure that the information is correct. Additionally, foresight into the information that is required later in the progression is available from the beginning. This allows users to organize and anticipate the work required to complete the progression. Users who are new to the system now have the ability to reduce the unknown aspects of the system.

3.2.2 Immediate Updates

Direct editing of the accumulated data is available while enacting a progression. A user can change information at any time without having to go backward in the progression and forfeit the later information. This also allows the user to keep track of their placement within the progression. Updates may not be allowed for some information items as the constraints of the system must be upheld. Nonetheless,

flexibility to make direct changes to the information already accumulated is afforded.

3.2.3 Historical Review

A user has access to the progression history. History includes the progression scenes, as well as the transaction information. The user can benefit from the ability to change, replay, or reuse historical information. Changing the history allows the user to move backward in the progression to undo actions. Replaying a progression may be useful for learning how progressions were previously completed by others; remembering what the user did last time they went through the progression; or for the supervisor to look at the work that an employee has performed. New or infrequent users would find the most benefit from this interaction. It is also beneficial for lengthy progressions, to view work that is not easily remembered. By saving the history of the progression, partial progressions can be closed and returned to at a later time, or parts of saved progressions can be reused in future progressions. This is useful when the user is interrupted during a session before they can complete the progression. Also, when work requirements are more ambiguous, this allows the user to explore different progression scenarios.

3.2.4 Multiple Progression Comparison

Multiple progressions can be used to process transactions concurrently. The ability to duplicate and/or view more than one progression at the same time allows for easy comparisons without having to lose work that is already completed, such as when trying out different scenarios or outcomes. The user can go through one possibility, then without losing that information try out another scenario. The outcomes can be

considered in a side–by–side manner.

3.2.5 Progression Batching

Progressions can be applied to multiple items to enable the user to perform a progression and have it affect more than one selected item in the system. For example, a user can perform a progression to change an employee's salary, but have it apply to ten employees. This is beneficial for saving time and consistency while managing large amounts of information.

3.2.6 Scene Reordering

Reordering the scenes of a progression allows the user set up the scenes as they would prefer to accomplish the tasks. This can help in making adjustments for changes in the business processes. Also users can put off work, which they cannot perform or are waiting on external information for, until a later time. This allows them to accomplish as much work as they can at a given time.

3.3 Summary

In this chapter we have introduced the intent of the progression model. We defined each of the parts that make up the progression model. Then we examined some of the benefits of utilizing the progression model. In the next chapter, we will define the progression notation and demonstrate how to specify the progression model.

Chapter 4

Progression Notation

The previous chapter introduced the progression model. In this chapter, we will define the progression notation for specifying a progression. In the next chapter, we will describe the progression analyzer, a prototype system we built to display information about a progression, interact with a progression, and manipulate progressions.

We have defined the progression notation based on the progression model for describing progressions [61]. We used XML syntax to ensure that it is XML-compliant. Based on the progression model described in Chapter 3, we have designed the progression notation for specifying workflow progressions. We use XML syntax to ensure that the notation is syntactically compatible with current research in XML-compliant user interface description languages. By developing an XML-compliant notation we can more easily share the results of our research and utilize the research of others when addressing issues beyond the scope of this thesis, such as user interface layout.

XML (Extensible Markup Language) is a text format markup language that was originally designed for electronic publishing, but is used increasingly for data exchange, and as we have seen in Chapter 2, for specifying user interfaces. There are two specifications involved in using XML: the XML document and the XML schema. An XML schema has been defined to structure the progression notation markup document, and throughout this thesis a progression notation markup docu-

ment will be referred to simply as the markup document. The next section provides a brief overview of XML. Subsequent sections describe how XML is used to specify progressions.

4.1 XML Markup Language

The XML document consists of a hierarchical structure of opening and closing tags. For example, an email document might be described in XML as:

```
<email>
    <from> jen@usask.ca </from>
    <to> dave@usask.ca </to>
    <time> 5:00pm </time>
    <subject> Hi </subject>
    <body> How are you? </body>
</email>
```

Each opening tag must have a corresponding closing tag. As well, the tags must be well-formed hierarchically. To determine whether an XML file is valid, it must follow the structure set out in the associated XML schema.

The XML schema describes the structure of the XML document. We have used the XML Schema Definition (XSD), but there are other schema types such as Document Type Definition (DTD). The XML schema is defined with elements, attributes, and text. For example, the corresponding XML schema for the email XML example could be as follows.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="email">
         <xs:complexType>
              <xs:sequence>
                   <xs:element name="from" type="xs:string" />
                   <xs:element name="to" type="xs:string" />
                   <xs:element name="time" type="xs:time" />
```

```
                <xs:element name="subject" type="xs:string" />
                <xs:element name="body" type="xs:string" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

The `xs:element` tag indicates an element; the attributes are specified within the tag, such as `name="email"`; and text appears in the XML specification, such as `jen@usask.ca`. Each element has a `maxOccurs` and a `minOccurs` attribute, which are defaulted to 1. The `xs:sequence` indicator specifies that the child elements must appear in a specific order. The `xs:choice` indicator specifies that only one of the children can be used in the specification.

Each element is either a simple element `xs:simpleType` or a complex element `xs:complexType`. A simple element contains only text; it has a `name` attribute and a `type` attribute. The type of a simple element can be either `xs:string`, `xs:decimal`, `xs:integer`, `xs:Boolean`, `xs:date`, `xs:time`, which are XML primitive types or a custom type. The `xs:restriction base="xs:string"` restricts the simple element type to `xs:string`. A complex element can contain a combination of other elements, attributes, and text. The attribute of `xs:schema`, `xmlns:xs="http://www.w3.org/2001/XMLSchema"` indicates that the elements and data types described in the XML schema, such as `schema`, `element`, `complexType`, `sequence`, and `string` come from the `"http://www.w3.org/2001/XMLSchema"` namespace. As well, these elements and data types should be prefixed with `xs:`.

4.2 Specifying Progressions with XML

The next sections describe the XML elements we use for describing progressions. Each element is described along with its schema, an example, and an informal description of the semantics. We will hierarchically go through each part of a progression. The XML elements specific to progressions are: progression, progressionMeta, progressionName, progressionTime, progressionID, transaction, transactionMeta, transactionName, transactionID, transactionSchema, scene, sceneMeta, sceneName, sceneTime, sceneID, userInterface, widgetGroup, widget, Jlabel, JTextField, Jbutton, workflow, status, worker, state, prev, next, constraints, reorder, before, and after. The elements are based on the progression model, but meta information has been added to each so as to identify and timestamp the enaction of the progression. There are always two schemas that describe a progression: the progression schema and the transaction schema. The progression schema outlines the entire progression. The transaction schema is separate because it is dependent on the information that is accumulated in the workflow transaction for the particular system domain.

The user interface definitions are based on the Java Programming language widgets that are used in the travel expense approval system example. The user interface definitions could be replaced with a User Interface Description language to allow for a more complex user interface. We chose to create this part of the notation to ensure that restrictions were not placed on our model or research that resulted from the structure of a predefined User Interface Description Language. Although our specification is limited, it provides the facility required for our example and is conducive

to extend to any Java widget.

4.3 Progression Element

A progression is the root of the hierarchical structure of the progression specification. It describes the transaction that is being built up, the scene information, and progression meta data.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="progression">
     <xs:complexType>
        <xs:sequence>
           <xs:element name="progressionMeta"> ... </xs:element>
           <xs:element name="transaction"> ... </xs:element>
           <xs:element name="scene" minOccurs="0" maxOccurs="unbounded">
           ... </xs:element>
        </xs:sequence>
     </xs:complexType>
  </xs:element>
</xs:schema>
```

A `<progression>` is the root element of the progression specification. The `<progression>` element is a complex element containing the following complex elements:

- `<progressionMeta>` is a complex element that identifies the progression.
- `<transaction>` is a complex element that specifies the workflow transaction for the progression.
- `<scene>` is a complex element that specifies a scene within a progression. It can occur zero or more times.

The following notation outlines a progression example. The schema for a progression can be found in Appendix A.

```
<progression>
    <progressionMeta> ... </progressionMeta>
    <transaction> ... </transaction>
    <scene> ... </scene>
</progression>
```

4.3.1 Progression Meta Element

A progressionMeta is the meta data that describes a progression.

```
...
<xs:element name="progressionMeta">
  <xs:complexType>
     <xs:sequence>
        <xs:element name="progressionName">
           <xs:simpleType>
              <xs:restriction base="xs:string"/>
           </xs:simpleType>
        </xs:element>
        <xs:element name="progressionTime">
           <xs:simpleType>
              <xs:restriction base="xs:date"/>
           </xs:simpleType>
        </xs:element>
        <xs:element name="progressionID">
           <xs:simpleType>
              <xs:restriction base="xs:string"/>
           </xs:simpleType>
        </xs:element>
     </xs:sequence>
  </xs:complexType>
</xs:element>
...
```

A `<progressionMeta>` is a complex element containing the following simple elements:

- `<progressionName>` is a simple element of type xs:string that would be meaningful to the user to identify the progression.

- <progressionTime> is a simple element of type xs:date that indicated the date that the progression was started on.

- <progressionID> is a simple element of type xs:string that uniquely identifies the progression.

The following notation outlines a progressionMeta example.

```
<progressionMeta>
    <progressionName> Expense Form 1 </progressionName>
    <progressionTime> 2004-11-01T14:11:54 </progressionTime>
    <progressionID> 128 </progressionID>
</progressionMeta>
```

4.3.2 Transaction Element

A transaction models the accumulation of information as actions are performed within a scene. It is displayed as the real world object, such as an invoice slip or an employee raise form. An XML schema is defined for the transaction as the structure might differ according to the type of object it is intended to represent. The structure of the information may be a series of keys and values or more complex with multiple levels.

```
...
<xs:element name="transaction">
  <xs:complexType>
     <xs:sequence>
        <xs:element name="transactionMeta"> ... </xs:element>
        <xs:element name="transactionSchema">
           <xs:simpleType>
               <xs:restriction base="xs:string"/>
           </xs:simpleType>
        </xs:element>
     </xs:sequence>
  </xs:complexType>
</xs:element>
...
```

A <transaction> is a complex element containing the following elements:

- <transactionMeta> is a complex element that identifies the workflow transaction.

- <transactionSchema> is a simple element of type xs:string that is the file name of the XML schema describing the transaction data.

The following notation outlines a transaction example.

```
<transaction>
    <transactionMeta> ... </transactionMeta>
    <transactionSchema> transaction.xsd </transactionSchema>
</transaction>
```

The schema element contains the file name of the XML schema that outlines the structure of the transaction that is required for the domain of the system. Each element in the schema has a name to connect it to the user interface widgets. The user can directly manipulate the transaction within the constraints of the system. For the example used in this thesis, the XML schema has three complex elements: contact, expenses, and account.

Transaction Meta Element

A transactionMeta is the meta data that describes a transaction.

```
...
<xs:element name="transactionMeta">
  <xs:complexType>
     <xs:sequence>
        <xs:element name="transactionID">
           <xs:simpleType>
              <xs:restriction base="xs:string"/>
           </xs:simpleType>
        </xs:element>
```

```
    </xs:sequence>
  </xs:complexType>
</xs:element>
...
```

A <transactionMeta> is a complex element containing the following element:

- A <transactionID> is a simple element of type xs:string that uniquely identifies the transaction from other transactions derived from the same XML schema. The first three digits indicate the progression that the transaction is associated with, and the final three digits indicate the individual transaction.

The following notation outlines a transactionMeta example. Figure 4.1 shows a screenshot of the result of the transaction specification.

```
<transactionMeta>
    <transactionID> 128 001 </transactionID>
</transactionMeta>
```

Transaction Schema

Each progression is associated with a data description of the workflow transaction. The workflow transaction differs for each application and is defined by the application architect. The example used in this thesis is for a *travel expense approval system*. A travel expense workflow transaction is made up of contact, expense, and account information. The schema will be explained further in Chapter 6 - Example and the complete schema can be found in Appendix B.

4.3.3 Scene Element

A scene is a step in the progression that indicates the user interface and workflow information.

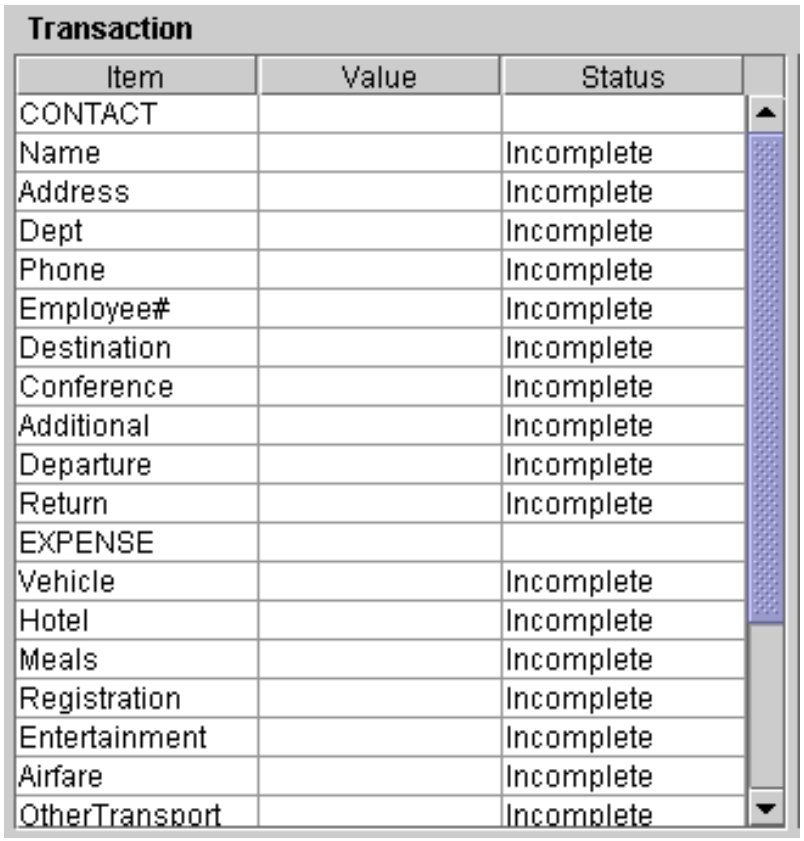

Figure 4.1: The transaction resulting from the transaction specification.

```
...
<xs:element name="scene" maxOccurs="unbounded">
  <xs:complexType>
     <xs:sequence>
        <xs:element name="sceneMeta"> ... </xs:element>
        <xs:element name="userInterface"> ... </xs:element>
        <xs:element name="workflow"> ... </xs:element>
     </xs:sequence>
  </xs:complexType>
</xs:element>
...
```

A `<scene>` is a complex element containing the following elements:

- `<sceneMeta>` is a complex element that identifies a scene.
- `<userInterface>` is a complex element that describes the user interface.

- `<workflow>` is a complex element that describes the workflow status and constraints.

The following notation outlines a scene example.

```
<scene>
    <sceneMeta> ... </sceneMeta>
    <userInterface> ... </userInterface>
    <workflow> ... </workflow>
</scene>
```

Scene Meta Element

A sceneMeta is the meta data that describes a scene.

```
...
<xs:element name="sceneMeta">
  <xs:complexType>
     <xs:sequence>
        <xs:element name="sceneName">
           <xs:simpleType>
              <xs:restriction base="xs:string"/>
           </xs:simpleType>
        </xs:element>
        <xs:element name="sceneTime">
           <xs:simpleType>
              <xs:restriction base="xs:date"/>
           </xs:simpleType>
        </xs:element>
        <xs:element name="sceneID">
           <xs:simpleType>
              <xs:restriction base="xs:string"/>
           </xs:simpleType>
        </xs:element>
     </xs:sequence>
  </xs:complexType>
</xs:element>
...
```

A `<sceneMeta>` is a complex element containing the following elements:

- `<sceneName>` is a simple element of type xs:string that is meaningful to the user to describe the scene.

- `<sceneTime>` is a simple element of type xs:date that indicates the date that the scene was started on

- `<sceneID>` is a simple element of type xs:string that uniquely identifies the scene. The first three numbers indicate the progression that it is associated with, and the last three uniquely identify the scene.

The `<sceneName>` is the name of the scene that should be meaningful to the user. The `<sceneTime>` indicates when the scene was initially accessed and worked on by the user. The `<sceneID>` is a combination of the `progressionID` followed by the scene number.

The following notation outlines a sceneMeta example.

```
<sceneMeta>
    <sceneName>Expense Form 1</sceneName>
    <sceneTime>3:00:00pm</sceneTime>
    <sceneID> 128 001 </sceneID>
</sceneMeta>
```

User Interface Element

A user interface specifies the structure of the user interface elements that are presented to the user for that scene. The user interface elements are connected to the transaction elements by name, such that an update to the user interface will cause the appropriate update to the transaction. Additionally, there is a connection to the workflow that indicates when actions within a scene are in progression and when all

actions are complete. Future extensions of this research may require a more complex connection between the user interface, transaction, and workflow.

We have defined the user interface using Java widgets. The `<userInterface>` element is a complex element that contains zero or more `<widgetGroup>` complex elements. A `<widgetGroup>` complex element is made up of zero or more `<widget>` complex elements.

```
...
<xs:element name="userInterface">
  <xs:complexType>
     <xs:sequence>
        <xs:element name="widgetGroup" maxOccurs="unbounded"> ...
        </xs:element>
     </xs:sequence>
  </xs:complexType>
</xs:element>
...
```

A `<userInterface>` is a complex element containing the following element:

- `<widgetGroup>` is a complex element that contains widgets that should be laid out together in the user interface.

Widget Group Element

A widgetGroup is a group of widgets that are related and should appear together in the user interface layout. A widget element represents a widget component, such as a text field.

```
...
<xs:element name="widgetGroup" maxOccurs="unbounded">
  <xs:complexType>
     <xs:sequence>
        <xs:element name="widget" maxOccurs="unbounded">
           <xs:complexType>
```

```
            <xs:choice>
               <xs:element name="JLabel">
                  <xs:simpleType>
                     <xs:restriction base="xs:string"/>
                  </xs:simpleType>
               </xs:element>
               <xs:element name="JTextField">
                  <xs:complexType mixed="true">
                     <xs:attribute name="isEditable">
                        <xs:simpleType>
                           <xs:restriction base="xs:boolean" />
                        </xs:simpleType>
                     </xs:attribute>
                  </xs:complexType>
               </xs:element>
               <xs:element name="JButton">
                  <xs:simpleType>
                     <xs:restriction base="xs:string"/>
                  </xs:simpleType>
               </xs:element>
            </xs:choice>
         </xs:complexType>
      </xs:element>
   </xs:sequence>
 </xs:complexType>
</xs:element>
...
```

A `<widget>` is a complex element containing the choice of the following elements:

- `<JLabel>` is a simple element of type xs:string that is the value of the JLabel.
- `<JTextField>` is a simple element of type xs:string that is the value of the JTextField.
- `<JButton>` is a simple element of type xs:string that is the value of the JButton.

The widget element has a choice indicator for the children. This means that for each widget element, a JLabel, JTextField, or JButton is specified. Each of the

children elements of widget also have an associated value attribute. The JTextField element has an attribute called "isEditable" which indicates whether or not the text field is editable.

The following notation outlines a widgetGroup example.

```
<widgetGroup>
     <widget>
          <JLabel value="First Name"> jlFirstName </JLabel>
     </widget>
     <widget>
          <JTextField value=" "> jtfFirstName </JTextField>
     </widget>
</widgetGroup>
```

We have only provided facility for simple user interfaces as our research has focused on the workflow concepts. There are many UIDLs that could be utilized in the user interface to provide a more aesthetic and user-friendly interface, such as UIML [1].

Workflow Element

A workflow describes workflow status and constraints. The scene's status indicates the current order in which the scenes are executed. There is one worker assigned to each scene. As the worker performs and completes tasks within the scene, the status of each scene is updated. The workflow constraints indicate any restrictions placed in the ordering of the scenes.

```
...
<xs:element name="workflow">
  <xs:complexType>
     <xs:sequence>
        <xs:element name="status"> ... </xs:element>
        <xs:element name="constraints"> ... </xs:element>
     </xs:sequence>
```

```
  </xs:complexType>
</xs:element>
...
```

A <workflow> is a complex element containing the following elements:

- <status> is a complex element that describes the workflow status in terms of recommended scene path, assigned worker, previous scene, and next scene.
- <constraints> is a complex element that describes the scene ordering constraints for the progression.

Status A status element indicates the recommended scene path, the worker assigned to each scene, and the status of each scene.

```
...
<xs:element name="status">
  <xs:complexType>
     <xs:sequence>
        <xs:element name="worker">
           <xs:simpleType>
              <xs:restriction base="xs:string"/>
           </xs:simpleType>
        </xs:element>
        <xs:element name="state">
           <xs:simpleType>
              <xs:restriction base="xs:string">
                 <xs:enumeration value=" Inactive "/>
                 <xs:enumeration value="Incomplete"/>
                 <xs:enumeration value="Complete"/>
                 <xs:enumeration value="InProgress"/>
              </xs:restriction>
           </xs:simpleType>
        </xs:element>
        <xs:element name="next" minOccurs="0">
           <xs:simpleType>
              <xs:restriction base="xs:string"/>
           </xs:simpleType>
        </xs:element>
```

```
        <xs:element name="prev" minOccurs="0">
           <xs:simpleType>
              <xs:restriction base="xs:string"/>
           </xs:simpleType>
            </xs:element>
     </xs:sequence>
  </xs:complexType>
</xs:element>
...
```

A <status> is a complex element containing the following elements:

- <worker> is a simple element of type xs:string that indicates the worker assigned to the scene.
- <state> is a simple element of type xs:string that is either Inactive, Incomplete, Complete, or In Progress.
- <next> is a simple element of type xs:string that indicates the scene that is ordered next according to the recommended scene path.
- <prev> is a simple element of type xs:string that indicates the scene that is ordered previously according to the enacted progression.

The following notation shows an example of a status specification. Figure 4.2 shows a rendering of the workflow status.

```
<status>
    <worker> claimant </worker>
    <state> inactive </state>
    <next> 3 </next>
    <prev> 1 </prev>
</status>
```

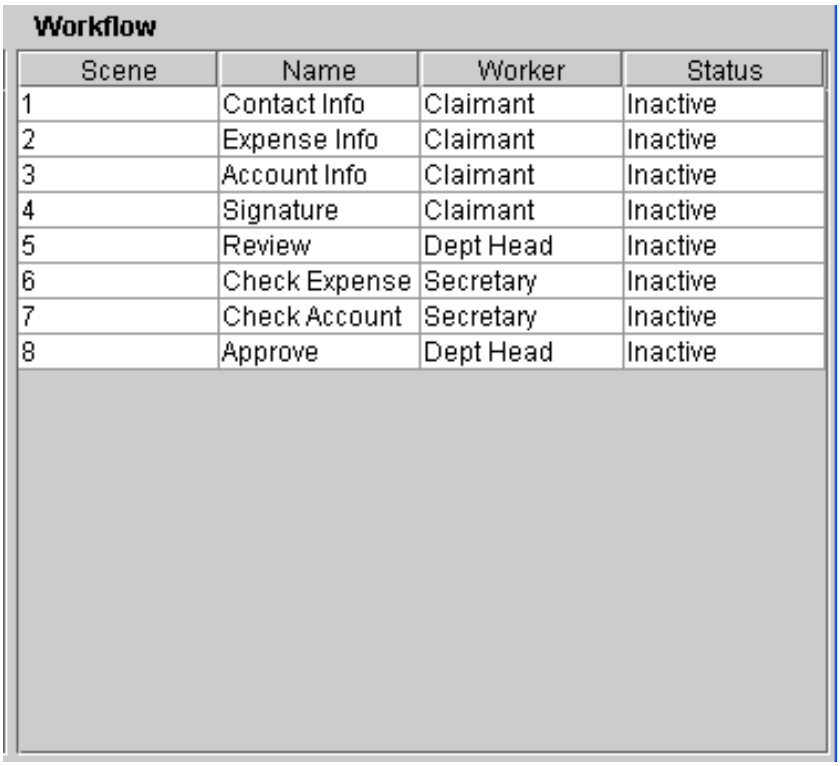

Figure 4.2: A rendering of the workflow panel according to the status specification.

Constraints

The constraints element is made up of reorder constraints that indicate the scene orderings that can disrupt the logical flow of work. For example, it is logical that the claimant would enter her account information before the scene where the secretary checks that the accounts are correct. However, there might be a case where the claimant only partially completes the account information, yet would like the secretary to check the account information at that time. Therefore, constraints are put in place to guide the users, yet flexibility is allowed to contend with exceptional situations.

```
...
<xs:element name="constraints">
  <xs:complexType>
     <xs:sequence>
```

```
        <xs:element name="reorder">
           <xs:complexType>
              <xs:sequence>
                 <xs:element name="before" maxOccurs="unbounded">
                    <xs:simpleType>
                       <xs:restriction base="xs:string"/>
                    </xs:simpleType>
                 </xs:element>
                 <xs:element name="after" maxOccurs="unbounded">
                    <xs:simpleType>
                       <xs:restriction base="xs:string"/>
                    </xs:simpleType>
                 </xs:element>
              </xs:sequence>
           </xs:complexType>
        </xs:element>
     </xs:sequence>
  </xs:complexType>
</xs:element>
...
```

A `<constraints>` is a complex element containing the following element:

- `<reorder>` is a complex element that describes whether a scene can be ordered after or before the current scene.

A `<reorder>` is a complex element containing the following elements:

- `<before>` is a simple element of type xs:string that indicates a scene that cannot be ordered before the current according to workflow constraints.
- `<after>` is a simple element of type xs:string that indicates a scene that cannot be ordered after the current according to workflow constraints.

For the scene that the constraints are specified in, the before and after elements identify which scene number can be ordered before the current scene and the scene

numbers that can be ordered after. The following notation shows an example of constraints that might occur in scene 2 with values provided for each child element.

```
<constraints>
    <reorder>
         <before> 5 </before>
         <before> 8 </before>
         <after> 1 </after>
    </reorder>
</constraints>
```

Incorporating actions into the notation requires some future research. Actions are intended to allow the model to interact with the application with workflow transactions rather than with fine-grain events. The user interacts with the user interface, which sends events to the workflow transaction and workflow status. Therefore, since all the events are handled at that level, the application can just be sent the workflow transaction for validation. Therefore, increased flexibility is provided because each event that occurs in the user interface does not need to be validated with the application to continue moving on in the workflow. The information resulting from the events is captured in the workflow transaction and can be validated periodically without interrupting the progress of the workflow until the final submission is ready to be made.

In this chapter we hierarchically went through each part of a progression and defined how it would be specified. The notation is a XML-based style, which is guided by the progression notation schema outlines at the end of the chapter. In the next chapter, we will discuss the progression analyzer prototype system we built to display information about a progression.

Chapter 5

Progression Analyzer Prototype

The previous chapter defined the progression notation for specifying a progression. In this chapter we introduce the progression analyzer, a prototype we built to execute and display information about a progression. We present a high-level description of the prototype system. Next, we will outline how the user can interact with the system. We will look at the transformation process that occurs with the markup document as the user interacts with the system. Finally, we will identify our flexibility goals that are achieved through the prototype system. In the next chapter, we will go through an extended example to demonstrate the progression analyzer.

5.1 Prototype Overview

The progression analyzer is a prototype system for displaying information about a progression, interacting with a progression, and manipulating a progression. The progression model is specified in progression notation to create the markup document. The progression analyzer reads in the progression markup document, displays the current scene information to the user, and updates the markup document according to the user's actions. The progression analyzer was created using the Java programming language and utilizes Java Swing for the user interface components.

The progression analyzer is divided into two sections: single and batch. The single progression section allows the user to interact with the user interface, transaction,

and workflow of one progression. The batch of multiple progressions allows the user to make a change and apply it to the transaction in each of the progressions within the batch.

The progression model is displayed to the user in the single progression section through four main panels:

- User Interface. This panel displays the widgets or interface elements that are specified in the user interface element of the markup document for the current scene.

- Transaction. This panel displays all the transaction items of the progression as specified in the workflow transaction element of the markup document, including the information that the user has entered up to that point in the progression. The structure of the transaction depends on the domain. It could be a series of keys and values or a more complex structure, such as a tree.

- Workflow. This panel displays the status information for the progression as specified in the workflow element of the markup document. The workflow status shows the scenes of the progression in the recommended order. Each scene is associated with a name, the worker assigned to complete it, and the current status of its completion; these are all displayed in a table format.

- Feedback. This panel consists of constraints, information status, and markup document. The constraints sub-panel displays any information related to constraints that are violated throughout the progression. The information status

sub-panel displays the information that is missing and required to complete the transaction. The markup document sub-panel displays the current state of the markup document.

The single section also has buttons to start a new progression, open an existing progression, save the current progression, and quit the progression analyzer.

The batch section displays buttons to create a new batch, add a progression to the current batch, delete a progression from the current batch, and quit the progression analyzer. A table displays the transaction information for each progression in the batch. The user can apply a change to one or more progressions by selecting the item to change, the progressions to apply the change, and the new value to use.

The progression analyzer facilitates visualization and interaction with the progression model. The progression analyzer is intended to interpret the progression notation from the markup document and display the information to the user. It handles getting the information from the user and applying it appropriately to the progression model. The user can see the parts of the progression model and apply actions to interact with or manipulate the progression. The user can perform three different actions:

- User actions. These are performed within the user interface panel and usually result in adding information to the transaction.
- Transaction actions. These are performed within the transaction panel and allow the user to directly edit any part of the transaction at any point in the progression.

- Workflow actions. These allow the user to manipulate the progression; they consist of reordering scenes, batching progressions, saving a progression, opening a progression, and traversing through any scene path.

5.1.1 Single Progression Example

An example of a simple progression in the progression analyzer is entering contact information. The first scene is Enter Name, the second scene is Enter City, and the third scene is Enter Phone Number. Figure 5.1 shows the progression analyzer displaying the first scene. The user interface panel shows a text field to enter the first name and a text field to enter the last name. The transaction panel shows all the information items that are included in the progression: first name, last name, city, and phone number; the values are empty at the beginning of the progression. The workflow panel shows the recommended scene path: 1, 2, 3; the name of each scene; the user that is assigned to perform each scene; and the current status of each scene. The first scene is displayed to the user, so the current status of scene 1 is **In Progress**. Then the user performs user actions to enter first name, enter last name, and press the submit button. The inputted information is updated in the transaction by filling in the values for first name and last name with the user's input.

The second scene is displayed to the user as shown in Figure 5.2. The user interface panel shows a text field to enter a city. The transaction panel shows the first name and last name values filled in from scene 1. The workflow panel shows that the status for the first scene is **Complete** and the status for the second scene is **In Progress**. Then the user performs user actions to enter the city and press the

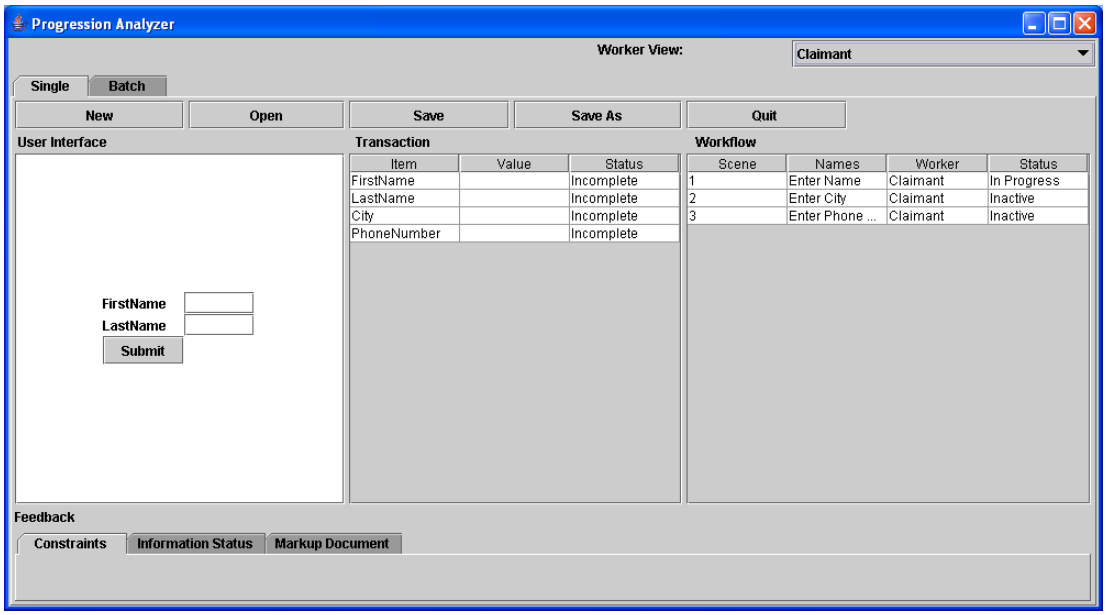

Figure 5.1: The progression analyzer displaying the Enter Name scene.

submit button. The inputted information is updated in the transaction by filling in the value for city with the user's input.

The third scene is displayed to the user as shown in Figure 5.3. The user interface panel shows a text field to enter a phone number. The transaction panel shows the first name and last name values filled in from scene 1, as well as the city value filled in from scene 2. The workflow panel shows the status for the first scene as **Complete**, the status for the second scene as **Complete**, and the status for the third scene as **In Progress**. Then the user performs user actions to enter the phone number and press the select button. The inputted information is updated in the transaction by filling in the value for phone number with the user's input.

At any time throughout the progression, the user can perform a transaction action. For instance, the user might realize that their phone number was entered incorrectly; therefore, the user could edit the text field within the transaction panel

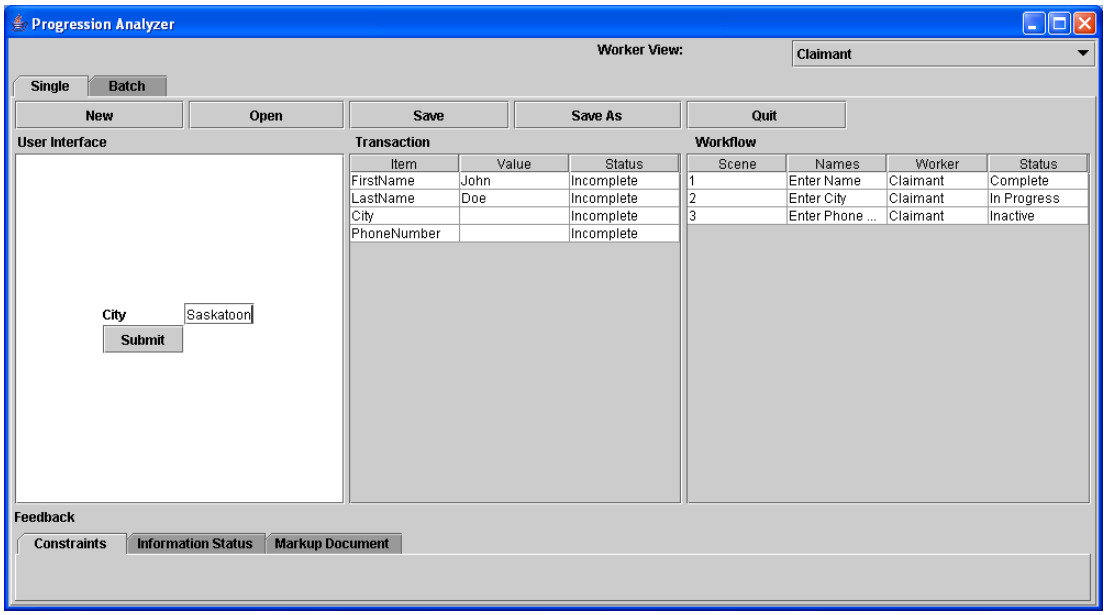

Figure 5.2: The progression analyzer displaying the Select City scene.

to make the correction. The user can also perform workflow actions. For example, if the user wants to reorder the recommended path, they can drag the scene rows into the preferred order.

5.1.2 Batch Progressions Example

An example of making an update to a batch of progressions can be seen in Figure 5.4. The progressions are similar to the one depicted in the previous example for entering contact information. The transaction consists of a first name, last name, city, and phone number. The user can select the item to update from the combo box. Next, the progressions within the batch to apply the changes are selected, in this example all progressions are selected. Finally, the new value is entered into the value text field. In this example, the user updates all the phone numbers to 306-966-5555.

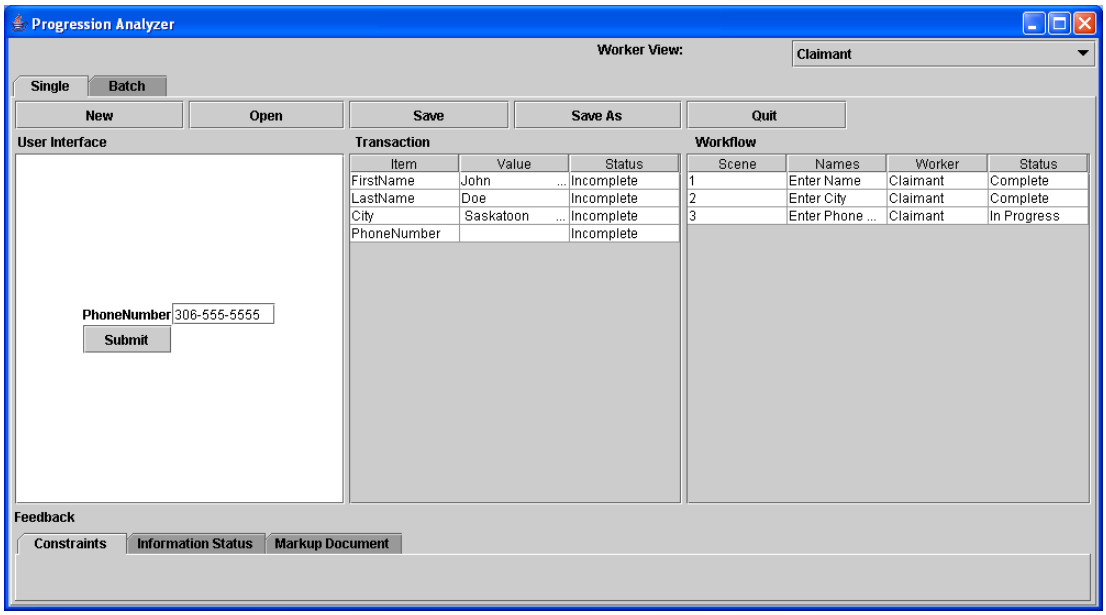

Figure 5.3: The progression analyzer displaying the Enter Phone Number scene.

5.2 Transformation

When the user performs any type of action within the single progression section of the progression analyzer, the input is taken and used to transform the markup document. For instance, in scene 1 of the single progression example, the markup document initially starts with no values filled in. Next, the user enters the first name, John, and the last name, Doe, in the user interface panel and presses the submit button. "John" and "Doe" are taken and used to fill in the values of the firstname JTextField and the lastname JTextField in the userInterface element of the markup document. The input is also used to fill in the value of the firstname JTextField and the lastname JTextField of the transaction element of the markup document. The transaction and workflow status are also updated appropriately. Currently, this is achieved through naming conventions, in future research we would have a more

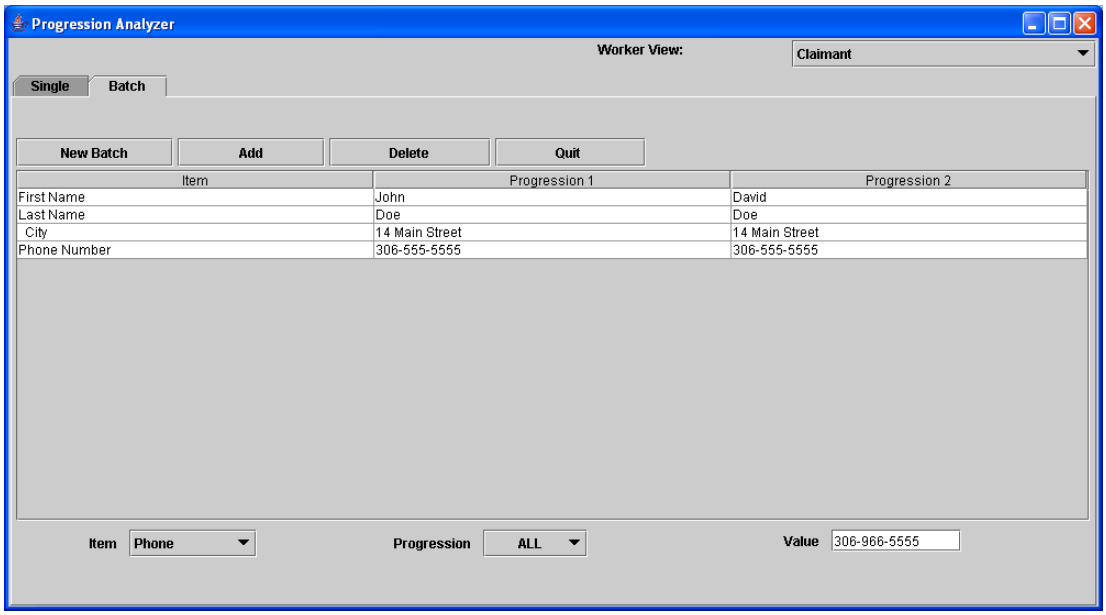

Figure 5.4: An example of applying a change to a batch of two progressions.

sophisticated binding. The following shows a section of the markup document before the transformation and after the transformation.

```
...
<userInterface>
    ...
    <JTextField value=" "> firstName </JTextField>
    <JTextField value=" "> lastName </JTextField>
    ...
<\userInterface>
...
<transaction>
    ...
    <firstName> </firstName>
    <lastName> </lastName>
    ...
</transaction>
...
```

In this version, the value of the text fields are " ". Also there is no text for firstname and lastname in the transaction. Then the user enters "John" as the first name and "Doe" as the last name.

```
...
<userInterface>
    ...
    <JTextField value="John"> firstName </JTextField>
    <JTextField value="Doe"> lastName </JTextField>
    ...
<\userInterface>
...
<transaction>
    ...
    <firstName> John </firstName>
    <lastName> Doe </lastName>
    ...
</transaction>
...
```

After the transformation, the values of the text fields become John and Doe, respectively. Also the text is added to the transaction.

The transformed markup document is then displayed to the user showing the next scene. The transaction panel shows that the first name is "John" and the last name is "Doe". If the user were to navigate to the first scene at this point, the text fields would appear with "John" and "Doe" as the values. Figure 5.5 shows the progression analyzer in scene 1 before the transformation. Figure 5.6 shows the progression analyzer in scene 2 after the transformation.

When the user performs an update to a batch of progressions, the markup document for each of the progressions is updated. The item is matched by naming conventions within the user interface element and the transaction element of the markup document. Then the new value is added or replaces the current value.

The markup document is also transformed when it is validated with the application. We would prefer to build up the transaction throughout the progression

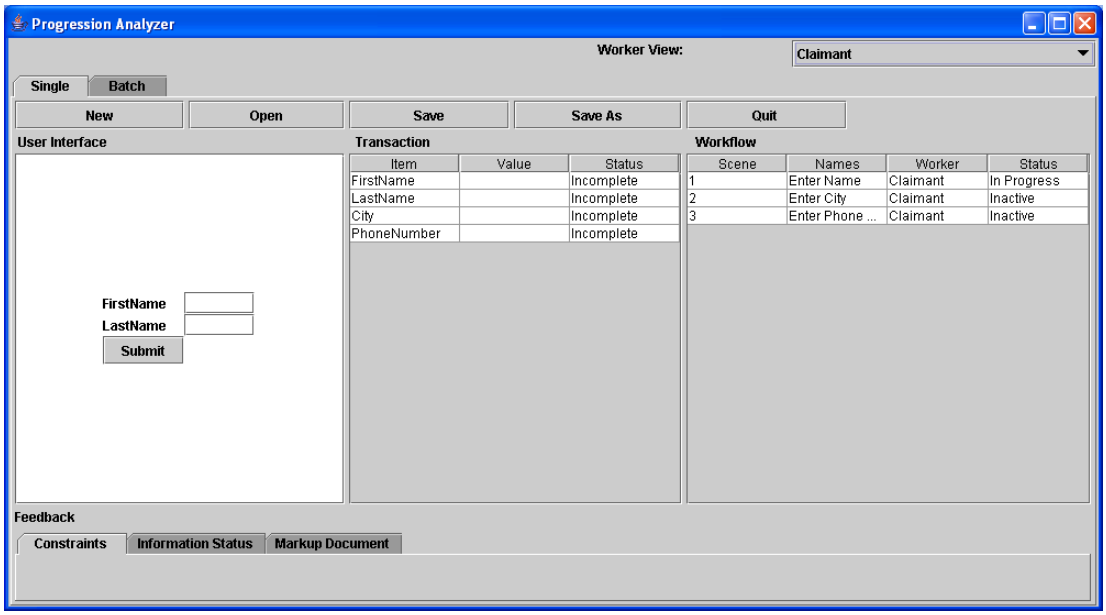

Figure 5.5: Scene 1 - Before the transformation.

and then validate it with the application when it is complete, to allow for the most flexibility. However, there are some cases where validation from the application is required during a progression; therefore validation may occur during the progression. An example of a validation with the application is if the user performs a transaction action to enter the city into the transaction for scene 2. The user interface panel provides a text field to enter a city. The user might enter a city that is not recognized by the system. Therefore, the value of the city is checked with the database of the application. If the city does not exist, the user is notified that the city value must be valid to complete the progression.

5.3 User Interaction

The user interacts with the user interface panel as in a typical interactive system by performing user actions, such as entering information into text fields, selecting from combo boxes, and so on. This is shown in Figure 5.7 as the user enters their contact

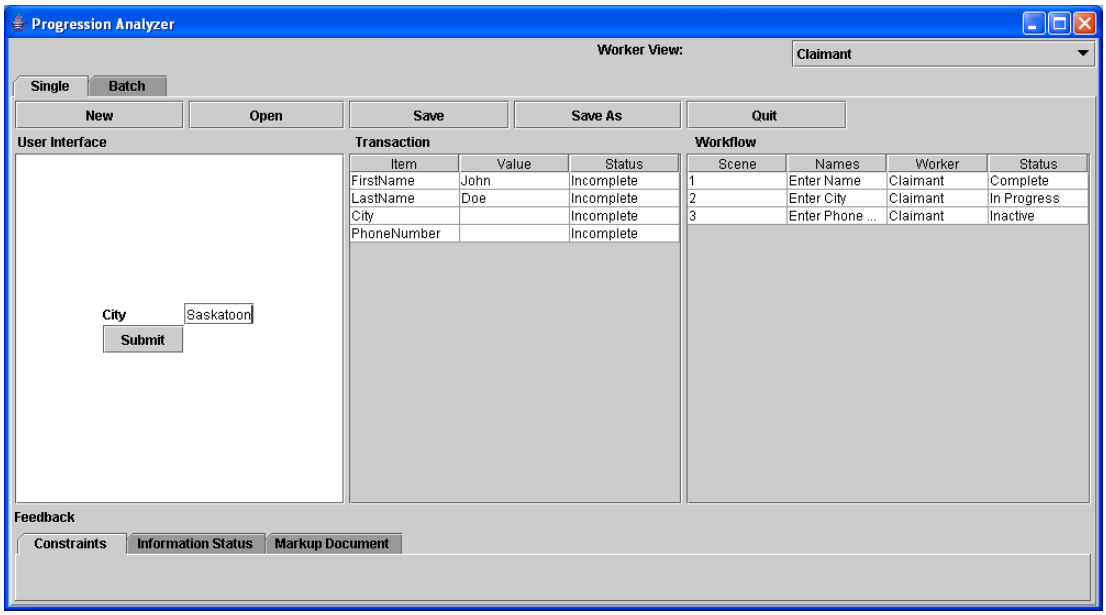

Figure 5.6: Scene 2 - After the transformation.

information and presses submit. Within the transaction panel, the user can perform transaction actions by directly editing the text fields to update the transaction, as shown in Figure 5.8. The user can also perform workflow actions to manipulate the progression, such as clicking on scene rows in the workflow panel to navigate to different scenes within the progression; dragging scene rows into different positions to reorder the scene path; saving the current state of the progression; and recalling saved progressions.

To navigate through the progression by the recommended path, the user interacts with the user interface panel and is restricted to the specified order of scenes. Alternatively, the user can navigate to a particular scene by selecting the row of that scene in the workflow panel. As shown in Figure 5.9, when scene 3 is selected in the workflow panel, the user interface for that scene is displayed in the user interface panel. Either way the user can traverse to another scene without completing all the

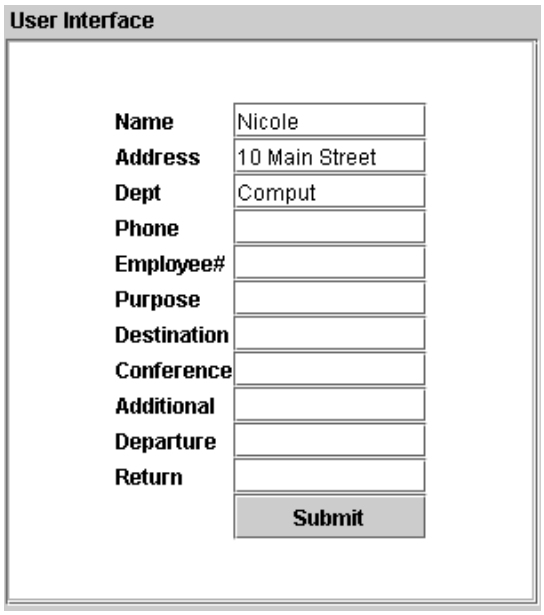

Figure 5.7: The user interface specification is displayed to the user and interacted with through the user interface panel of the progression analyzer.

required information in the previous scene. Additionally, any transaction updates from the previous scene are reflected in the transaction panel.

The user can also rearrange the initial recommended scene path as outlined in the workflow panel. The recommended scene path indicates the order that the scenes will be displayed if the user interacts solely with the user interface panel to navigate. This is done by dragging a scene, represented as a table row, into the desired order from the beginning top position to the ending bottom position. The workflow section of the markup document outlines the ordering constraints that are required by the system. If any constraints are violated in the reordering, the feedback is displayed to the user in the **Constraints** tab. Figure 5.10 shows the violated constraints displayed back to the user in the **Constraints** tab.

The user can save the current state of the markup document through the **Save**

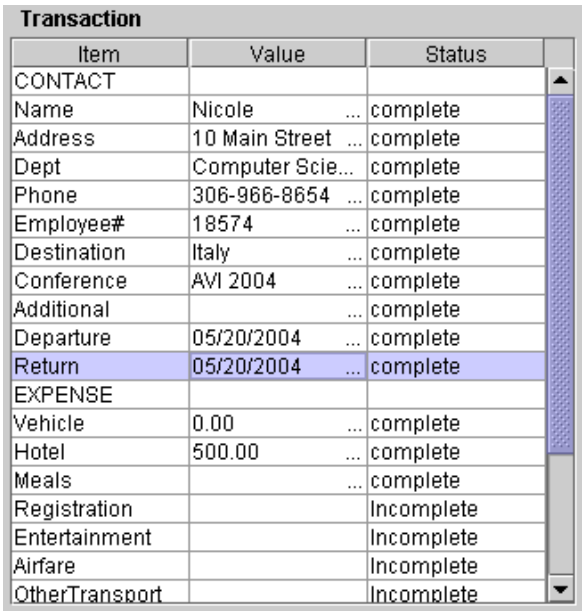

Figure 5.8: The information elements within the transaction can be directly edited to change a value, such as the Return date of the trip.

button. A markup document can be opened by the progression analyzer with the **Open** button. Therefore, versioning can be used to allow the user to go back to the progression at an earlier time, even if the current document has since changed.

5.4 Flexibility Goals

We identified flexible workflow as allowing the user to keep track of data as it is accumulated, reorder the steps taken to complete the work, pass work on to another worker to be completed, and view the completed work as well as the work that remains to be completed. Therefore, to provide this flexibility, our prototype has been designed to support the following requirements:

Task Reordering. We have provided support to allow a single user to reorder the scene within a progression by rearranging the scenes in the workflow panel. The new ordering is checked with the workflow constraints to ensure that it does not

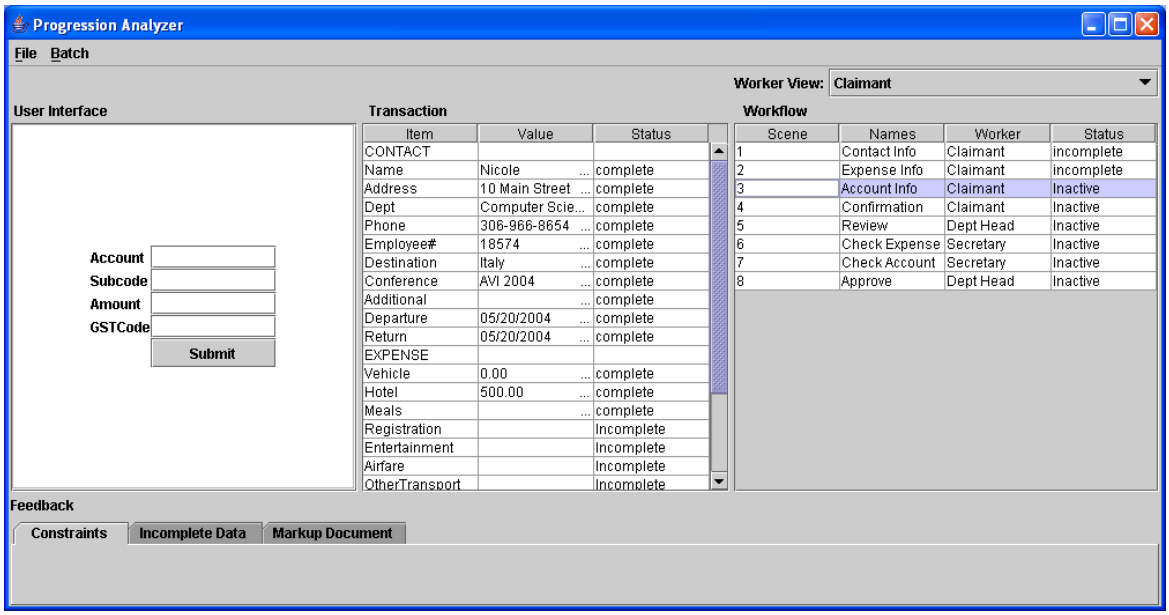

Figure 5.9: The progression analyzer shows that when a scene is selected in the workflow panel, the corresponding user interface is displayed in the user interface panel.

violate ordering requirements. However, even if the constraints are violated, the ability for the user to continue on is maintained.

Batching. We have provided support to make changes to multiple progressions or transactions from one workflow action. Batching allows the user to select a number of progressions that are partially or fully completed and apply a single user action or a series of user actions.

Error Feedback. We have provided support for feedback to the user about constraint violation and invalidations with the application. When workflow constraints are violated, the user is presented with information about what violations were made. Also, when checking for validation with the system, the user is sent back an indication on how the information was not valid.

Information Status. We have provided support for information status identi-

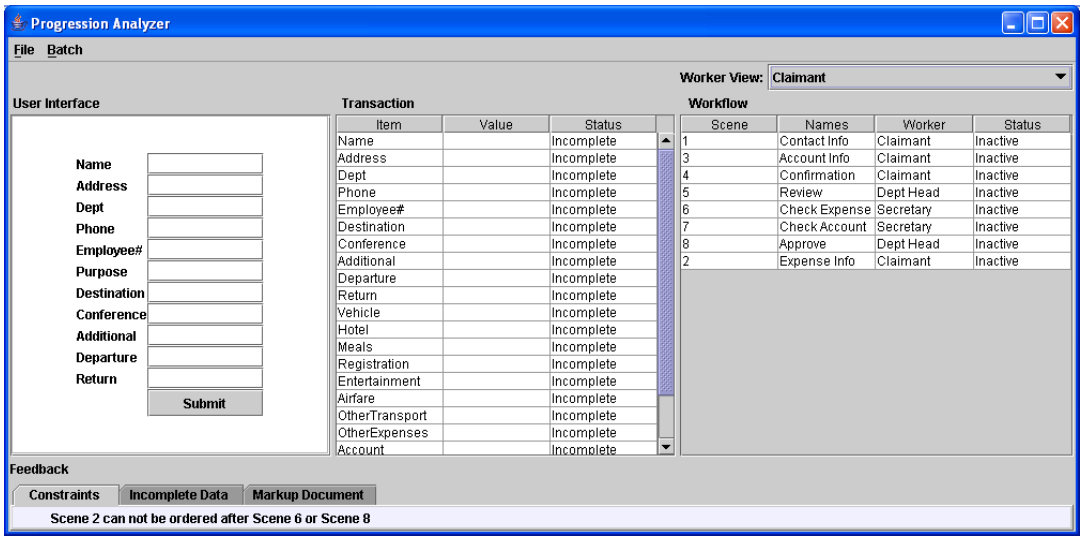

Figure 5.10: By reordering the second scene to the last position, reorder constraints are violated for this system. The expense information must be entered before it can be checked or approved.

fication. Allowing different information states assists in providing additional system flexibility while maintaining the system information requirements. We intend to indicate to the user whether transaction information is complete, incomplete, stale, or timed-out. Also, we look at workflow status information as being inactive, incomplete, complete, and in-progress.

5.5 Prototype Limitations

The progression analyzer system has some limitations. The current version of the system does not fully parse and update a complex XML structured markup document. This is feasible to implement, however, through utilizing Java DOM parser components. The application core in this system does not include a complete backend as would be expected in a commercial system. We simulate the application to show that the connection to the application can occur at the transaction level of granularity. Also, in the current version of the progression analyzer, the transaction

panel does not represent a travel expense form to our preferred likeness. This is in part due to the rendering sophistication, as well as to assist in screenshot readability. More time and research would allow for more complicated and aesthetically pleasant user interface rendering.

In the current chapter, we introduced the progression analyzer prototype system. We looked at the types of interactions afforded to the user for performing and manipulating progressions. Then we went through the transformation process that the markup document undertakes. We examined the flexibility goals that we set out to achieve. Finally, we mentioned some assumptions that were made when building the system. In the next chapter, we will look at the travel expense approval system example.

Chapter 6

Travel Expense Approval Example

The last chapter introduced the progression analyzer prototype system that displays information about the progression. In this chapter we will outline the travel expense approval system that we will use to demonstrate our progression model and prototype system. We will begin by outlining the typical scenario of the scenes in the progression. Next, we will describe the types of flexibility that we intend to demonstrate. We will then describe the scenarios that we developed to test these types of flexibility.

A proof of concept case study has been performed to test the progression analyzer prototype with the progression model. The domain of the case study is a system for processing travel expense claim forms. There is a single claimant filling out a form for one trip, which must be approved by the department head, checked by the secretary, and administered from payment services. Figure 6.1 shows the UML 2.0 activity diagram for this workflow.

6.1 Domain

The scenario for a typical progression in this travel expense claim form system is outlined as follows:

- *Scene 1: Enter contact information into expense form (claimant)*

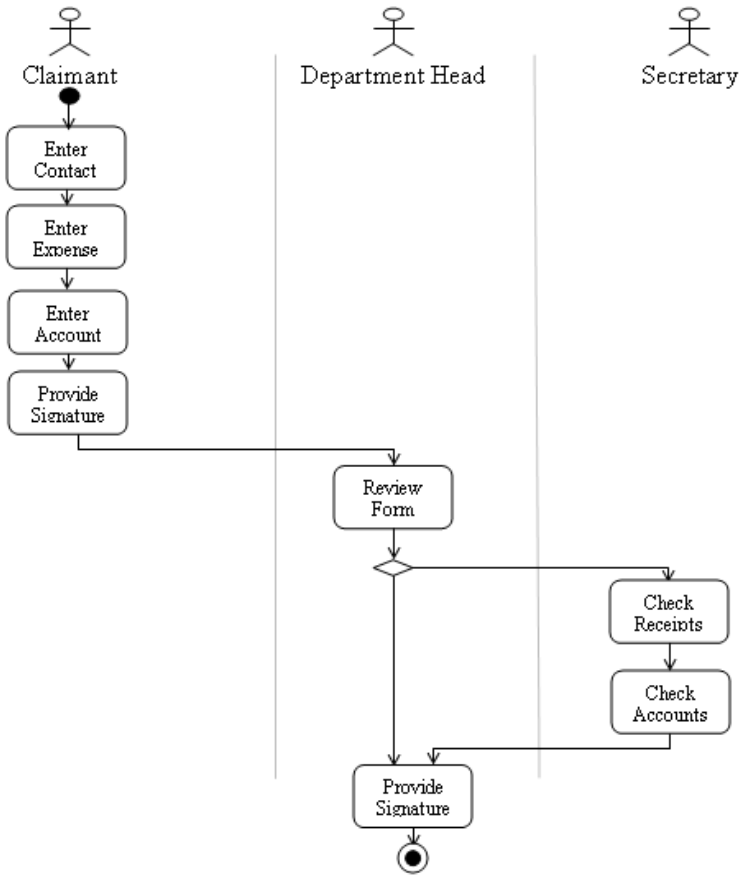

Figure 6.1: A UML 2.0 activity diagram for the expense approval workflow process.

- The claimant is prompted to enter her contact information including name, address, department, phone number, employee number, purpose, destination, conference, departure date, arrival date, and any additional information.

- *Scene 2: Enter expense information into expense form (claimant)*

  - The claimant is prompted to enter her applicable expense information including the amounts for vehicle, hotel, meals, registration, entertainment, airfare, other transportation, and other expenses.

- *Scene 3: Enter account information into expense form (claimant)*
  - The claimant is prompted to enter her account information including the account code, subaccount code, amount to charge to that account, and the gstCode if applicable for each account involved in the expense form.
- *Scene 4: Provide claimant confirmation (claimant)*
  - The claimant is presented with the contact, expense, and account information that was inputted and prompted to confirm that the information is correct.
- *Scene 5: Review expense form (department head)*
  - The department head is presented with the contact, expense, and account information that was entered by the claimant and prompted to review the information. Then he can choose to have the secretary check the information or go to the final approval.
- *Scene 6: Checks receipts (secretary)*
  - The secretary receives the expense information and is prompted to review and okay the expenses.
- *Scene 7: Checks account codes (secretary)*
  - The secretary receives the account information and is prompted to review and okay the account information.
- *Scene 8: Provides approval (department head)*

– The department head is presented with the contact, expense, and account information. He can see whether or not the information was okayed by the secretary. Then he is prompted to approve the expense reimbursement.

6.1.1 Transaction Schema

The workflow transaction for this example outlines the contact, expense, and account information that is required to complete the progression. We will describe the transaction schema that is referred to in the markup document. The following section describes the transaction for this particular example. The transaction XML schema contains the following complex elements:

- `<contact>` is a complex element that describes the contact information that the user inputs.
- `<expenses>` is a complex element that describes the travel expenses that the user inputs.
- `<accounts>` is a complex element that describes the account information that the user inputs. It can occur one or more times.

Contact

Contact holds the claimant's contact information as the travel expense form is filled out.

```
...
<xs:element name="contact">
  <xs:complexType>
     <xs:sequence>
        <xs:element name="name">
```

```
            <xs:complexType mixed="true">
               <xs:attribute name="status" use="required">
                  <xs:simpleType>
                     <xs:restriction base="xs:string"/>
                  </xs:simpleType>
               </xs:attribute>
            </xs:complexType>
         </xs:element>
         <xs:element name="address"> ... </xs:element>
         <xs:element name="department"> ... </xs:element>
         <xs:element name="phone"> ... </xs:element>
         <xs:element name="employeeNum"> ... </xs:element>
         <xs:element name="purpose" minOccurs="0"> ... </xs:element>
         <xs:element name="destination"> ... </xs:element>
         <xs:element name="conference"> ... </xs:element>
         <xs:element name="additional" minOccurs="0"> ... </xs:element>
         <xs:element name="departDate"> ... </xs:element>
         <xs:element name="returnDate"> ... </xs:element>
      </xs:sequence>
   </xs:complexType>
</xs:element>
...
```

A <contact> is a complex element made up of the following elements:

- <name> is a complex element that describes the name of the claimant.
- <address> is a complex element that describes the address of the claimant.
- <phone> is a complex element that describes the phone number of the claimant.
- <employeeNum> is a complex element that describes the employee number of the claimant.
- <purpose> is a complex element that describes the purpose of the claimant's travel.

- `<destination>` is a complex element that describes the destination of the claimant's travel.

- `<conference>` is a complex element that describes the conference that the claimant is traveling to.

- `<additional>` is a complex element that describes any additional information that the claimant wants to enter.

- `<departDate>` is a complex element that describes the departure date of the claimant's travel.

- `<returnDate>` is a complex element that describes the return date of the claimant's travel.

Each of the children of `contact` have an attribute called status, which indicates the status of that item's completion. The following notation shows an example of a contact specification.

```
<contact>
    <name status="incomplete">Nicole</name>
    <address status="incomplete">10 Main Street</address>
    <department status="incomplete">Computer Science</department>
    <phone status="incomplete">306-966-8654</phone>
    <employeeNum status="incomplete">18574</employeeNum>
    <purpose status="incomplete">Conference</purpose>
    <destination status="incomplete">Italy</destination>
    <conference status="incomplete">AVI 2004</conference>
    <additional status="incomplete"></additional>
    <departDate status="incomplete">05/20/2004</departDate>
    <returnDate status="incomplete">05/28/2004</returnDate>
</contact>
```

Expenses

The `<expenses>` elements hold the claimant's expense information as the travel expense form is filled out.

```
...
<xs:element name="contact">
  <xs:complexType>
     <xs:sequence>
        <xs:element name="vehicle" minOccurs="0">
           <xs:complexType mixed="true">
              <xs:attribute name="checked" use="required">
                 <xs:simpleType>
                    <xs:restriction base="xs:Boolean"/>
                 </xs:simpleType>
              </xs:attribute>
              <xs:attribute name="status" use="required">
                 <xs:simpleType>
                    <xs:restriction base="xs:string"/>
                 </xs:simpleType>
              </xs:attribute>
           </xs:complexType>
        </xs:element>
        <xs:element name="hotel" minOccurs="0"> ... </xs:element>
        <xs:element name="meals" minOccurs="0"> ... </xs:element>
        <xs:element name="registrationFee" minOccurs="0"> ... </xs:element>
        <xs:element name="entertainment" minOccurs="0"> ... </xs:element>
        <xs:element name="airFare" minOccurs="0"> ... </xs:element>
        <xs:element name="otherTransport" minOccurs="0"> ... </xs:element>
        <xs:element name="otherExpenses" minOccurs="0"> ... </xs:element>
        <xs:element name="total"> ... </xs:element>
     </xs:sequence>
  </xs:complexType>
</xs:element>
...
```

Expenses is a complex element made up of the following elements:

- `<vehicle>` is a complex element that describes the amount of vehicle expenses incurred during travel.

- `<hotel>` is a complex element that describes the amount of hotel expenses incurred during travel.
- `<meals>` is a complex element that describes the amount of meal expenses incurred during travel.
- `<entertainment>` is a complex element that describes the amount of entertainment expenses incurred during travel.
- `<airFare>` is a complex element that describes the cost of the airplane ticket.
- `<otherTransport>` is a complex element that describes the amount of other transportation expenses incurred during travel.
- `<otherExpenses>` is a complex element that describes the amount of other general expenses incurred during travel.
- `<total>` is a complex element that describes the total amount of expenses incurred during travel.

Each of the children elements of `expenses` has an attribute, checked, of type xs:Boolean and an attribute, status, of type xs:string. The attribute checked is defaulted as "false" and is changed to "true" to indicate that the information has been checked for accuracy. The following notation shows an example of a expenses specification.

```
<expenses>
    <vehicle checked="false" status="incomplete">$0.00
    </vehicle>
    <hotel checked="false" status="incomplete">$500.00
```

```
    </hotel>
    <meals checked="false" status="incomplete">$300.00
    </meals>
    <registrationFee checked="false" status="incomplete">$200.00
    </registrationFee>
    <entertainment checked="false" status="incomplete">$200.00
    </entertainment>
    <airFare checked="false" status="incomplete">$850.00
    </airFare>
    <otherTransport checked="false" status="incomplete">$0.00
    </otherTransport>
    <otherExpenses checked="false" status="incomplete">$50.00
    </otherExpenses>
</expenses>
```

Account

The account elements hold the claimant's account information as the travel expense form is filled out.

```
...
<xs:element name="account" minOccurs="0">
  <xs:complexType>
     <xs:sequence>
        <xs:element name="code">
           <xs:complexType mixed="true">
              <xs:attribute name="checked" use="required">
                 <xs:simpleType>
                    <xs:restriction base="xs:boolean"/>
                 </xs:simpleType>
              </xs:attribute>
              <xs:attribute name="status" use="required">
                 <xs:simpleType>
                    <xs:restriction base="xs:string"/>
                 </xs:simpleType>
              </xs:attribute>
           </xs:complexType>
        </xs:element>
        <xs:element name="subCode"> ... </xs:element>
        <xs:element name="amount"> ... </xs:element>
        <xs:element name="gstCode" minOccurs="0">... </xs:element>
```

```
    </xs:sequence>
  </xs:complexType>
</xs:element>

...
```

An <account> is a complex element made up of the following elements:

- <code> is a complex element that describes the company account code that should be used to charge the amount provided.
- <subCode> is a complex element that describes the subcode that should be used to charge the amount provided.
- <amount> is a complex element that describes the amount of money that should be charged to the provided account codes.
- <gstCode> is a complex element that describes the company GST code number.

Each of the children elements of account has an attribute, checked, of type xs:Boolean and an attribute, status, of type xs:string. The attribute checked is defaulted as "false" and is changed to "true" to indicate that the information has been checked for accuracy. The following notation is an example of an account specification. Figure 6.2 shows a screenshot of the transaction.

```
<account>
    <code checked="false" status="incomplete">4500</code>
    <subCode checked="false" status="incomplete">005</subCode>
    <amount checked="false" status="incomplete">$2100.00</amount>
    <gstCode checked="false" status="incomplete">321659875461</gstCode>
</account>
```

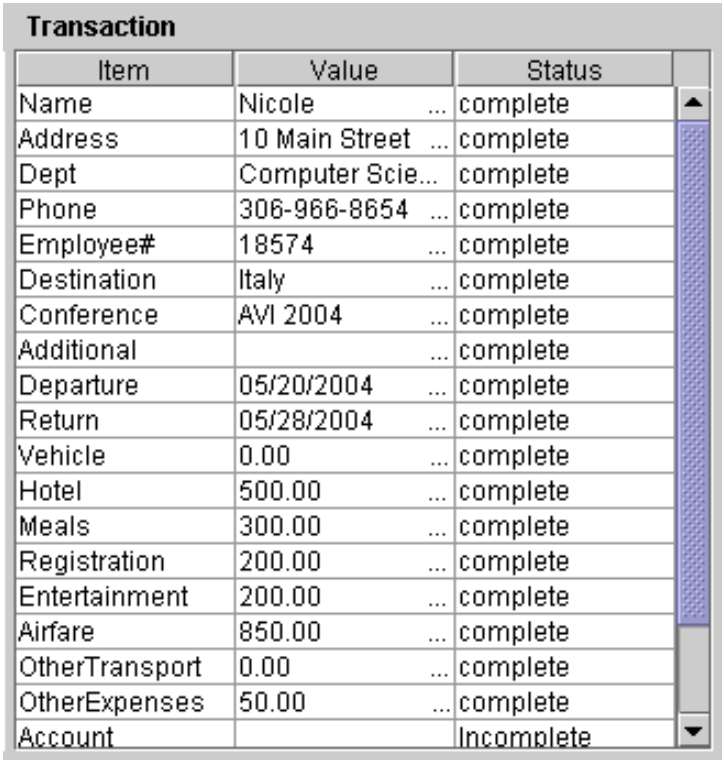

Figure 6.2: The transaction resulting from contact, expenses, and account.

6.1.2 Progression Description

In the first three scenes, the user interface is displayed that prompts the user for their contact information, expense information, and account information, respectively. A condensed version of the information elements required in scenes 1, 2, and 3 are provided in Figure 6.3.

After the user has completed the information entry, there is a prompt to review the input and confirm that it is correct. Upon confirmation, the transaction information is sent to the department head for review. The department head then reviews the information and sends it on to the secretary or sends it on for approval. The secretary checks the expenses and receipts in the sixth scene. Then in the sev-

Contact Information	Expense Information	Account Information
Name	Vehicle	Account Code
Address	Hotel	Subaccount Code
Department	Meals	Amount
Phone Number	Registration	GST Code
Employee Number	Entertainment	
Purpose	Airfare	
Destination	Other Transportation	
Conference	Other Expenses	
Departure Date		
Arrival Date		
Additional Information		

Figure 6.3: The expense form elements for contact, expenses, and accounts.

enth scene, ensures that the account information is correct. The transaction is sent back to the department head for final approval. The payroll services department processes and pays out the expenses. This scenario would be a common occurrence under predictable conditions. We intend to provide flexibility support to assist users in situations that are not as predictable.

6.2 Flexibility Goals

We identified flexible workflow as allowing the user to keep track of data as it is accumulated, reorder the steps they take to complete the work, pass work on to another worker to be completed, and view the completed work as well as the work that remains to be completed. Therefore, in our example, we are attempting to support the following requirements:

Task Reordering. By allowing the user to reorder the scenes within a progression, the user can perform tasks at her own discretion as the needs of the situation require. Workflow constraints are checked to ensure that ordering requirements are

not violated. However, if a violation occurs, the user should be notified with the option to continue on and fix the violation at a later time.

Batching. Batching allows the user to select a number of progressions that are partially or fully completed, and apply a user action or series of user actions to them. By providing support to allow the user to apply changes to a batch of progressions, the user can make multiple changes in one step. This saves time and repetition.

Error Feedback. This involves indicating to the user feedback about constraints that are violated and information that is invalid within the application. This allows the user to quickly identify the workflow violations and maintain the workflow constraints.

Information Status. By providing indication of the status of information associated with the transaction and workflow, the user can see what tasks have been worked on and what tasks need to be completed. This also indicates how the user's actions have impacted on the transaction and workflow constraints.

6.3 Scenarios

We have chosen seven scenarios for this example to test out the workflow flexibility goals that were outlined for the progression analyzer prototype. We next describe each scenario.

6.3.1 Single Worker Reorder

The claimant decides to switch around his individual activities to enter expense information, then account information, followed by contact information. The claimant is prompted to enter his contact information. Since he wants to bypass the contact

information at that time, he goes to the workflow panel and rearranges the scene rows. He drags scene 1 down to below scene 3 and clicks on scene 2, which is now top on the list. The new order is enter expense information, enter account information, and enter contact information. The expense information is displayed to the user in the user interface panel as it is now the current scene. The workflow constraints are checked to see if any reorder constraints are violated; they indicate which scenes can go before or after each other. Any violations are displayed in the constraints feedback tab. However, this scenario has not violated any constraints. The progression will now be enacted to the new order. Figure 6.4 shows the reordered scene path in the workflow panel.

Workflow

Scene	Names	Worker	Status
2	Expense Info	Claimant	Inactive
3	Account Info	Claimant	Inactive
1	Contact Info	Claimant	Inactive
4	Confirmation	Claimant	Inactive
5	Review	Dept Head	Inactive
6	Check Expense	Secretary	Inactive
7	Check Account	Secretary	Inactive
8	Approve	Dept Head	Inactive

Figure 6.4: The progression analyzer showing the reordered scenes.

The only change to the markup document is in the <next> element of the rearranged scenes. In scene 1, the <next> element is changed to 4. In scene 2, the <next> element remains as 3. In scene 3, the <next> element is changed to 1. The following progression notation example depicts the change to scene 3. The element <prev> remains empty until scene 3 is in progress, because it indicates the scene

that was visited previously during the progression execution.

```
<status>
    <meta>
         <sceneName>Account Info</sceneName>
         <time>2004-12-27T03:15:16</time>
         <sceneID> 128 001 </sceneID>
    </meta>
    <worker> claimant </worker>
    <state> inactive </state>
    <next> 1 </next>
    <prev> </prev>
    <complete> false </complete>
</status:scene>
```

6.3.2 Multiple Worker Reorder

The department head decides to have the secretary check the claimant's expenses and accounts before it is sent to her for review. She wants to reorder the tasks of the claimant and the secretary. The department head rearranges the scenes so that the contact information is entered; then the expense information; then the secretary checks the expense information; then the claimant enters the account information; then the secretary checks the account information; following that, the information is sent to the department head for review and approval. Since the department head has the correct permissions, she can change the order of tasks for the other workers. Figure 6.5 shows the new scene order in the workflow panel of the progression analyzer.

The only change occurs in the <status> element of the rearranged scenes. In this case, the department head accidently rearranges the scenes to have the secretary review the accounts before the scene where the claimant enters the accounts. Immediately, the constraint violation is displayed to the user. Seeing that it is a violation

Workflow

Scene	Names	Worker	Status
1	Contact Info	Claimant	Inactive
2	Expense Info	Claimant	Inactive
6	Check Expense	Secretary	Inactive
3	Account Info	Claimant	Inactive
7	Check Account	Secretary	Inactive
4	Confirmation	Claimant	Inactive
5	Review	Dept Head	Inactive
8	Approve	Dept Head	Inactive

Figure 6.5: The workflow panel showing the reordered scenes.

that the department head does not want to make, she switches the two scenes to repair the violation.

6.3.3 Batching

The company has been issued a new GST exempt code. Therefore, the department head would like to apply this change to all the pending travel expense forms. The department head selects the **Batch** tab. Then adds the two progressions that he would like to modify to the batch. He selects GST code from the list of items, chooses the apply the change to all progressions, enters the new GST code in the value text field. Figure 6.6 shows the Account Info scene where the GST code is changed during the batching.

For the transformation, the markup document element(s) that are changed by the user's action(s) are identified. Next, each file that was selected is parsed and the values of the changed element(s) are updated to the new values. Next, when a progression from the batch is reopened in the progression analyzer, the new GST code will be displayed to the user. The following progression notation example shows a specification of the account element with the new value for the GST code element.

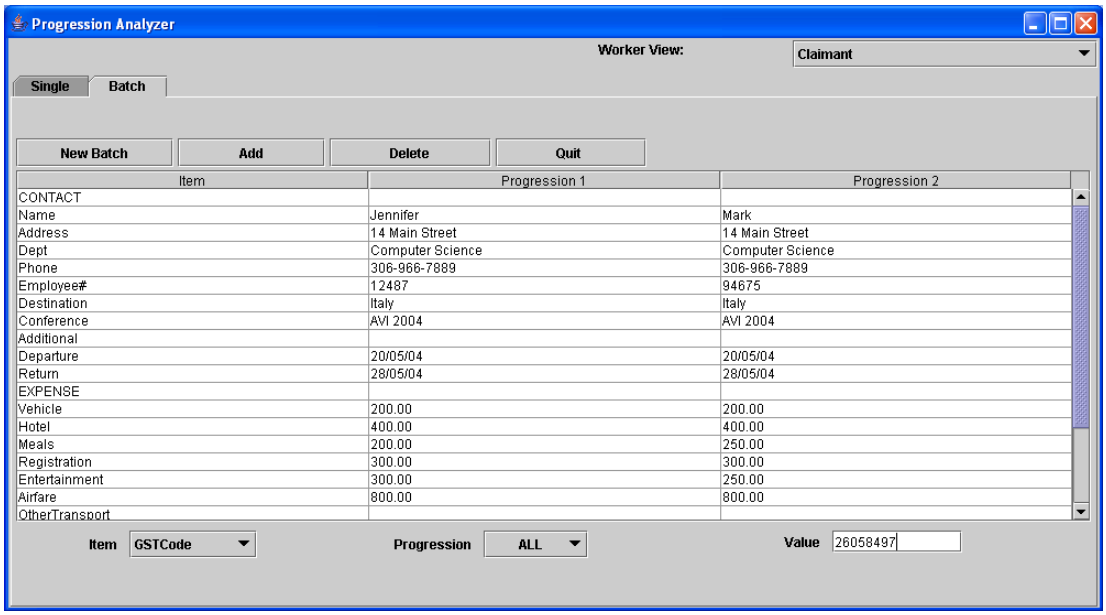

Figure 6.6: The progression analyzer during a batch update. The batch section allows the user to apply user action(s) to multiple progressions.

```
<account>
    <code>4500</code>
    <subCode>005</subCode>
    <amount>$2100.00</amount>
    <gstCode>321659875461</gstCode>
</account>
```

6.3.4 Constraint Feedback

The claimant decides to reorder the recommended scene path because she does not have all the account information that is required; however, she would like to complete the remaining scenes in the progression. She chooses to move scene 3, Account Info, to the last scene in the progression. Since approval can only be given once the account information is available for approval and the account information must be provided to check that it is correct, two ordering constraints are violated. These violations are presented to the user through the **Constraints** tab in the feedback panel. They

indicate which scenes are in violation. Figure 6.7 shows the **Constraints** tab in the feedback panel with the message that Scene 3, Account Info, must be ordered before Scene 7, Check Account, and Scene 8, Approve.

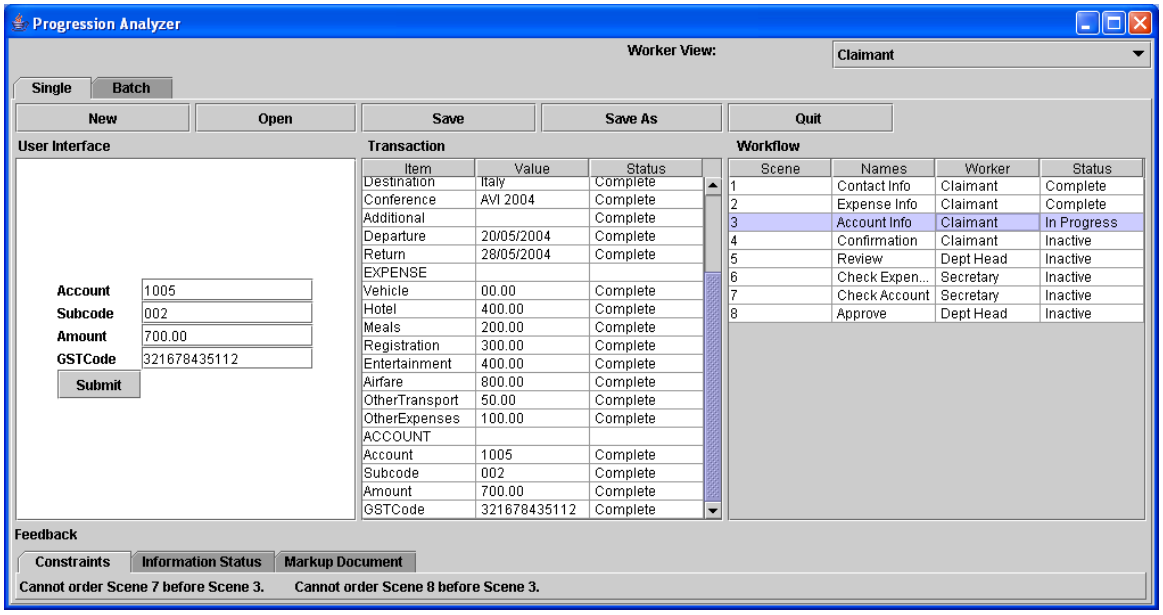

Figure 6.7: The progression analyzer displaying constraint violation feedback to the user. The Constraints tab of the progression analyzer indicates that Scene 3 cannot be ordered after Scene 7 or Scene 8.

The markup document is flagged after an error is identified when validating the user's input with the application core. In the account specification, the error is noted as indicated in the following progression notation example.

```
<workflow>
     <status>
          <worker> claimant </worker>
          <state> Inactive </state>
          <next> 4 </next>
          <prev> 2 </prev>
     </status>
     <constraints>
          <reorder>
               <before> 7
                    <errorType>Cannot order Scene 7 before Scene 3.
                    </errorType>
```

```
                </before>
                <before> 8
                     <errorType>Cannot order Scene 8 before Scene 3.
                     </errorType>
                </before>
           </reorder>
     </constraints>
</workflow>
```

6.3.5 Stale Data

After the department head has provided her final approval, the claimant realizes that he needs to make a change to his transportation expense. The claimant makes his edit, which causes the department head's approval to become stale. The claimant is notified that the approval is stale and the transaction cannot be completed without a new approval. Figure 6.8 indicates the stale status of the approval from the department head in the workflow panel.

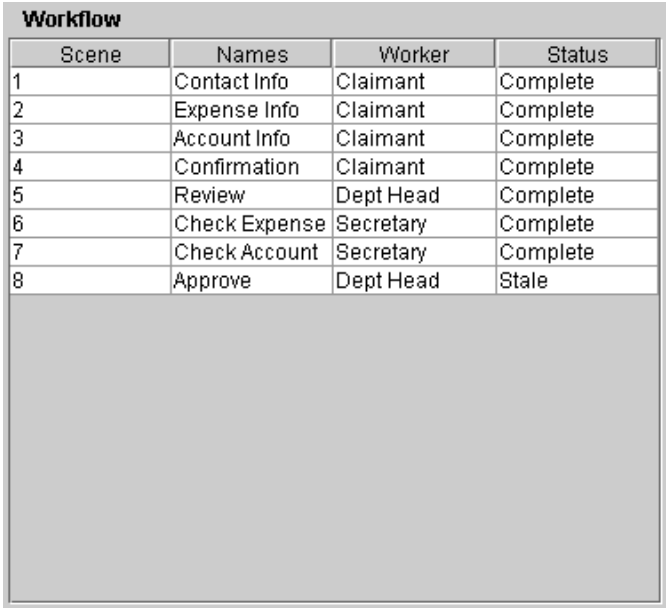

Figure 6.8: The workflow panel showing the stale scene status. The workflow panel in the progression analyzer shows that the approval of the travel expense form has become stale.

The workflow status in the markup document is also changed to reflect the stale data. The following progression notation example shows the status specification in the markup document.

```
<status>
    <meta>
         <sceneName>Approve</sceneName>
         <time>3:00:00pm</time>
         <sceneID> 128 001 </sceneID>
    </meta>
    <worker> claimant </worker>
    <state> stale </state>
    ...
</status>
```

6.3.6 Partial Data

The claimant wants to bypass some of the expense information and fill out all the personal and account information that he has available. Therefore, he would like to bypass some of the information fields. The information fields that are not complete are kept track of to notify the user upon request or when the user attempts to complete the progression. Figure 6.9 shows the transaction with the missing information elements and the corresponding list in the **Information Status** feedback tab.

The markup document keeps track of the status of each information element in the transaction. The following progression notation example shows the transaction specification indicating incomplete status of the employee number and phone number information elements.

```
<transaction>
    ...
    <contact>
         <name status= "complete">Nicole</name>
       ...
```

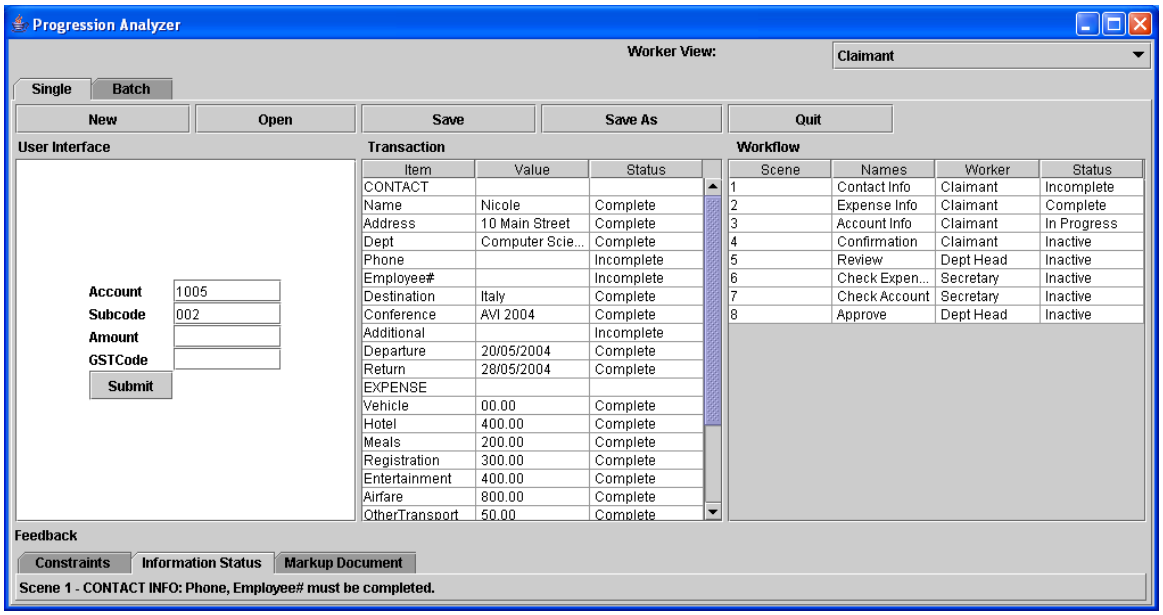

Figure 6.9: The progression analyzer showing incomplete status of elements in the transaction. The progression analyzer shows that the phone number and employee number information is incomplete in the transaction.

```
        <phoneNum status= "incomplete"></phoneNum>
        ...
          <employeeNum status= "incomplete"></employeeNum>
          <purpose status="complete">Conference</purpose>
          ...
    </contact>
...
</transaction>
```

6.3.7 Timed-out Data

The expense form is completed and submitted to payroll services. However, one of the accounts has been frozen. The cheque cannot be sent out until the account is unfrozen. After three days, the information is timed-out. Feedback is provided to the claimant indicating that an account has been frozen. The markup document is updated with the timed-out status of the account information in the transaction and workflow. The following progression notation example shows the transaction

specification with the timed-out status of the account element. Figure 6.10 shows the transaction panel indicating that the account information is timed-out.

```
<transaction>
    ...
    <account>
         <code status="timed-out">2507</code>
         <subcode status="timed-out">003</subcode>
         <amount status="timed-out">200.00</amount>
         <gstCode status="timed-out">32165468431</gstCode>
    </account>
    ...
</transaction>
```

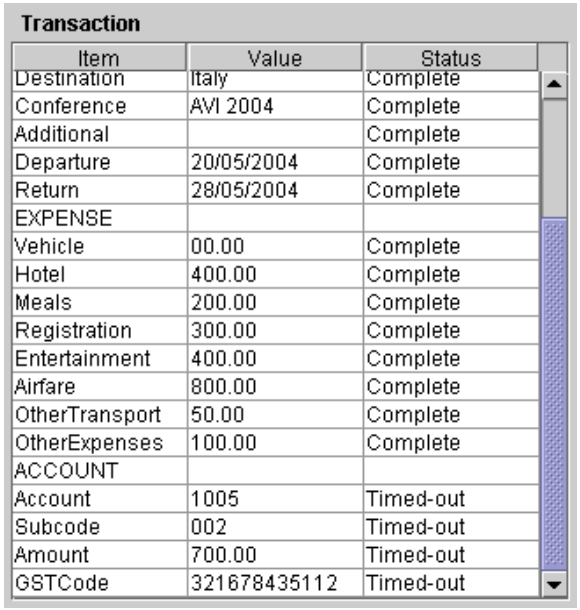

Figure 6.10: The transaction panel showing the timed-out account information. The progression analyzer provides feedback to the user under the **Information Status** tab indicating that the account has been frozen. The status of the account information elements are set to timed-out.

In this chapter we outlined the domain of the travel expense approval system example. Next, we revisited the flexibility goals that we intended to demonstrate with the example. Finally, we showed seven scenarios to indicate the flexibility goal

achievement. In the next chapter, we will summarize contributions, and show the direction for future work.

Chapter 7

Conclusion

This research has focused on supporting business process flexibility in transaction-based systems. We will summarize the motivation and contributions in the next section. In the second section will discuss the areas we have identified for future work, which includes integrating our progression notation with a developed user interface description language, explicitly specifying the user actions within the notation, defining the workflow constraints more completely, and developing a comprehensive system that could be used in a practical setting.

7.1 Summary

Business organizations adopt business processes, whether implicit or explicit, to organize work toward the successful completion of business goals. A business process is "a set of one or more linked activities that collectively realize a business goal, normally within the context of an organizational structure defining functional roles and relationships" [64]. Business processes help ensure that work is performed consistently and reliably. Interactive transaction-based software systems are created to automate part or all of the business process.

As the business environment often experiences changes due to many factors, business processes inherently change. Changes in business processes require changes in the software systems that provide support. Otherwise, users must contend with

inadequate software that does not meet their needs. Re-developing the software is time consuming and costly, which is infeasible for the rapid business process changes. Therefore, we have looked at designing the software with flexibility in mind to assist the user in dealing with business process changes.

To modularly design interactive transaction-based systems with additional flexibile functionality, we have looked to model-based user interface design. In Chapter 2 - Related Work we reviewed the current research in workflow, task models, use cases, software evolution, business process constraints, model-based user interface design, and user interface description languages. In Chapter 3 - Progression Model we developed our progression model that incorporates workflow concepts to support flexibility. We also suggested many possible benefits that the progression model could facilitate, such as information orientation, immediate updates, historical review, concurrent multiple progression comparison, progression batching, and scene reordering. We then focused on a smaller set of these benefits, which we termed our flexibility goals. They include:

- Task Reordering. Allowing the user to reorder the scenes within a progression enables the user to perform tasks as the needs of the situation require.

- Progression Batching. Batching enables the user to perform action(s) and apply them to multiple progression, to make consistent, fast updates.

- Error Feedback. This allows the user to quickly identify the workflow violations and invalid information when the transaction is submitted to the application core.

- Information Status Feedback. By displaying the information status of the transaction elements and workflow, the user can see what tasks have been worked on and what tasks need to be completed.

In Chapter 4 – Progression Notation we defined the progression notation to specify a progression. In Chapter 5 – Prototype we introduced our prototype system, the progression analyzer, to display information about a progression. Finally, in Chapter 6 – Travel Expense Approval Example we outlined a proof of concept case study of a travel expense approval system to demonstrate that our desired flexibility goals have been achieved. We intended to support task reordering, which was shown through the single worker reorder and multiple worker reorder example scenarios. The ability to apply actions to a batch of progressions was shown in the batching scenario. Providing enriched error feedback to the user, was shown in the error feedback scenario. Information status support was demonstrated in the stale data, partial data, and timed-out data scenarios.

Explicitly recording and manipulating progressions allows us to dynamically change the workflow of an interactive system. Benefits include, improving the plasticity of an interactive system (workflow plasticity), providing a coarser integration with an application (transaction-based integration), and additional workflow functionality. Our contributions that result from this research are:

- Developing a model that makes workflow status and workflow transactions explicit; and,

- Expressing workflow concepts in an XML-based notation.

Our secondary contributions are included in our model, but require further research to be incorporated into our notation and fully demonstrated in our prototype. The secondary contributions are as follows:

- Supporting workflow actions in information systems; and,
- Connecting the user interface to the application at a larger granularity level of a workflow transaction.

These all provide support for increased flexibility in the flow of work within information systems.

7.2 Future Work

There are some extensions and improvements to this research that we would like to pursue in the future. They include integrating our workflow notation concepts with a user interface description language from current research, explicitly specifying user actions within the notation to fully support a flexible interaction between the user interface and the application, defining workflow constraints to support more complicated system constraints, and developing a comprehensive system for practical use and further study and analysis.

7.2.1 Integrate with a UIDL

We would look at integrating our workflow concepts with a developed user interface description language, such as UIML [1]. This would involve utilizing that language for the user interface subsection of the progression model and adjusting the remainder of the model to a similar style. We chose to create a simple notation for describing

the user interface because we did not want to inadvertently restrict our workflow concepts with language constructs. The additional research of a more established language would also provide more sophisticated user interface rendering capabilities to present more complicated user interfaces.

7.2.2 Make User Actions Explicit

We would also like to incorporate the user actions into the progression notation. The user actions that we have examined are user interface, transaction, and workflow. User interface actions are interactions that the user performs to the user interface elements, such as a text field or select box. Transaction actions are interactions that the user performs to the transaction. Workflow actions update or manipulate the progression workflow. We would like to specify these actions and the results that occur within the system in the progression notation. This will help in making the actions explicit, rather than hidden within the system implementation. It will also facilitate a more complex connection of the updates between the user interface, transaction, and workflow status. Furthermore, this list of user actions is not comprehensive for all systems. Therefore, we would like to further explore and expand this list.

7.2.3 Define Workflow Constraints

We also put minimal focus on the workflow constraint definitions. These are important to ensure that some business and system rules are not violated. We would look to the Business Process Execution Language, which has XML-based constraint notation [3]. Our current model only includes ordering constraints. There are other

issues that may be important to the business goals of an organization, such as worker permissions for rearranging scenes. Therefore, we would like to further examine the potential types of constraints and how they can be specified.

7.2.4 Develop a Comprehensive System

We would ideally develop a more comprehensive system that includes all the functionality that would be needed in a practical working system. This would include a distributed system that handles multiple users and complete processing of the markup document. Although our model and notation considers multiple users, our prototype system does not demonstrate a realistic situation of a multi-user system. We would like to establish this further and determine new user actions and constraints that could benefit a multi-user system. We would also want to expand the user functionality to include more of the benefits that the progression model can facilitate, such as enacting multiple progressions, visually playing out progressions, and progression history manipulation. If we had a more complete system we could perform more in-depth user analysis to inform system redesign. This may include monitoring the user to determine their most efficient task path. Additionally, we would like to examine and support a more complexly structured transaction. The current system demonstrates a key and value structure. It would be beneficial to support other structures, such as a tree.

7.2.5 Discussion

One motivational problem still resides with certain types of information systems. This exists where companies benefit from the inflexibility of their system. It is

most common in systems where the user is a customer. For example, the company benefits from forcing the customer to enter their contact information first, so even if the customer does not complete their transaction the company retains the user information. Furthermore, a company may force all users to follow one specific process to act as a guide for users as well as ensure the company receives consistent results. Nonetheless, there are many customer user systems that could greatly benefit from the type of flexibility that we elicit from the progression model, and these systems would see an increase in user satisfaction.

7.3 Conclusion

Users require flexibility when interacting with information systems that are created to support business processes, because the business environment changes so rapidly. This thesis has focused on supporting flexibility within these systems by defining a workflow model, the progression model that makes the workflow status and workflow transaction explicit. It also looks at supporting user actions and connecting the user interface to the application through the workflow transaction. A notation has been designed to specify the progression model. To examine the model and notation, a prototype has been implemented. An example of a travel expense approval system has demonstrated that our model notation and prototype meet our flexibility goals of supporting task reordering, progression batching, constraint error feedback, and information status awareness.

References

[1] M. Abrams, C. Phanouriou, A.L. Batongbacal, S. Williams, and J. Shuster. UIML: An appliance-independent XML user interface language. In *Proceedings of 8th International World-Wide Web Conference WWWs*, pages 661–665, 1999.

[2] D. Amyot and G. Mussbacher. On the extension of UML with use case maps concepts. *<<UML>>2000*, pages 16–31, 2000.

[3] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services version 1.1, May 2003.

[4] J. Annett, K.D. Duncan, R.B. Stammers, and M.J. Gray. *Task Analysis*. London: Her Majesty's Stationery Office, 1971.

[5] R. Anzbock, S. Dustdar, and H. Gall. Title. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 649 – 656. ACM Press, 2002.

[6] L. Ardissono, A. Goy, and G. Petrone. Enabling conversations with web services. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 819 – 826. ACM Press, 2003.

[7] A. Arsanjani, D. Chamberlain, D. Gisolfi, R. Konuru, J. Macnaught, S. Maes, R. Merrick, D. Mundel, T.V. Raman, S. Ramaswamy, T. Schaeck, R. Thompson, A. Diaz, J. Lucassen, and C.F. Wiecha. Specification: (wsxl) web service experience language version 2, April 2002.

[8] L. Aversano and G. Canfora. Introducing eservices in business process models. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering table of contents*, pages 481 – 488. ACM Press, 2002.

[9] P. Azevedo, R. Merrick, and D. Roberts. OVID to AUIML user-oriented interface modeling. In *Proceedings of 1st International Workshop Towards a UML Profile for Interactive Systems Development TUPIS00*. York, October 2000.

[10] A. Bernstein. How can cooperative work tools support dynamic group processes? bridging the specificity frontier. *CSCW00*, pages 279–288, 2000.

[11] R. J. A. Buhr and R. S. Casselman. *Use case maps for object-oriented systems*. Prentice-Hall, Inc., 1996.

[12] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, N. Souchon, L. Bouillon, M. Florins, and J. Vanderdonckt. Plasticity of user interfaces: A revisited reference framework. In *Proceedings of 1st International Workshop on Task*

Models and Diagrams for user interface design TAMODIA'2002, pages 127–134, July 2002.

[13] A. Campi, E. Martinez, and P.S. Pietro. Experiences with a formal method for design and automatic checking of user interfaces. In *(IUI-CADUI 2004) Workshop: Making Model-based UI Design Practical: Usable and Open Methods and Tool*, January 2004.

[14] S.K. Card, T.P. Moran, and A. Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum, 1983.

[15] A. Cockburn. Structuring use cases with goals. journal of object-oriented programming. *CHANGE*, pages 35–40, September-October 1997.

[16] A. Cockburn. Structuring use cases with goals. journal of object-oriented programming. *CHANGE*, pages 56–62, November-December 1997.

[17] F. Curbera, Y. Goland, J. Klein, F. Leymann, S. Roller, S. Thatte, and S. Weerawarana. Business process execution language for web services, version 1.0, July 2002.

[18] J. Eisenstein and A.R. Puerta. Towards a general computational framework for model-based interface development systems. In *Proceedings of the 4th international conference on Intelligent user interfaces*, pages 171 – 178, 1998.

[19] J. Eisenstein, J. Vanderdonckt, and A.R. Puerta. Applying model-based techniques to the development of UIs for mobile computers. In *Intelligent User Interfaces*, pages 69–76, 2001.

[20] P. Forbrig, A. Dittmar, D. Reichart, and D. Sinnig. From models to interactive systems tool support and XIML. In *(IUI-CADUI 2004) Workshop: Making Model-based UI Design Practical: Usable and Open Methods and Tool*, January 2004.

[21] M. Frank and J. Foley. Model–based user interface design by example and by answering questions. In *Proceedings of INTERCHI, ACM Conference on Human Factors in Computing Systems*, pages 161–162, April 1993.

[22] D. Georgakopoulos, M. F. Hornick, and A. P. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, pages 119–153, 1995.

[23] J. Grudin and S.E. Poltrock. Computer-supported cooperative work and groupware. In *In M. Zelkowitz (Ed.), Advances in Computers*, volume 45, pages 269–320. Academic Press, 1997.

[24] J. Haake and W. Wang. Flexible support for business processes: Extending cooperative hypermedia with process support. *GROUP97*, pages 341–350, 1997.

[25] H.R. Hartson, A.C. Siochi, and D. Hix. The UAN: a user-oriented representation for direct manipulation interface designs. *ACM Transactions on Information*, pages 181–203, July 1990.

[26] P. Heinl, S. Horn, S. Jablonski, J. Neeb, K. Stein, and M.A. Teschke. A comprehensive approach to flexibility in workflow management systems. *WACC99*, pages 79–88, 1999.

[27] Vanderdonckt J., Limbourg Q., and Florins M. Synchronized model-based designof multiple user interfaces. In T. Radhakrishnan A. Seffah and G.Canals (Eds.), editors, *Proceedings of Workshop on Multiple User Interfaces over the Internet: Engineering and Applications Trends*, September 2001.

[28] P. Castells J.A. Macas. An eud approach for making mbui practical. In *(IUI-CADUI 2004) Workshop: Making Model-based UI Design Practical: Usable and Open Methods and Tool*, January 2004.

[29] B.E. John and D.E. Kieras. The GOMS family of user interface analysis techniques: comparison and contrast. *ACM Transactions on Computer-Human Interaction*, 3(4):320–351, 1996.

[30] D. Kieras. A guide to goms task analysis, 1994.

[31] K. Luyten, T. Clerckx, K. Coninx, and J. Vanderdonckt. Derivation of a dialog model from a task model by activity chain extraction. *DSV-IS2003*, 2003.

[32] K. Luyten and K. Coninx. An XML-based runtime user interface description language for mobile computing devices. In *Proceedings of the 8th International Workshop on Interactive Systems: Design, Specification, and Verification-Revised Papers*, pages 1–15. Springer-Verlag, 2001.

[33] K. Luyten, C. Vandervelpen, and K. Coninx. Adaptable user interfaces in component based development for embedded systems. In *Proceedings of the 9th Int. Workshop on Design, Specification, and Verification of Interactive Systems DSV-IS2002*. Springer Verlag, June 2002.

[34] K. Luyten, C. Vandervelpen, and K. Coninx. *Migratable User Interface Descriptions in Component-Based Development*, volume 2545, pages 62–76. Springer, 2002.

[35] P. Mangan and S. Sadiq. On building workflow models for flexible processes. In *Proceedings of the thirteenth Australasian conference on Database technologies*, volume 5, pages 103–109. Australian Computer Society, Inc., 2002.

[36] D.A. Manolescu. Workflow enactment with continuation and future objects. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA02*, pages 40–51, 2002.

[37] D.A. Manolescu. An extensible workflow architecture with objects and patterns. *Chapter 4 in Technology of Object-Oriented Languages, Systems, and Architectures*, 2003.

[38] P.J. Molina. A review to model-based user interface development technology. In *(IUI-CADUI 2004) Workshop: Making Model-based UI Design Practical: Usable and Open Methods and Tool*, January 2004.

[39] G. Mori, F. Paterno, and C. Santoro. Ctte: Support for developing and analyzing task models for interactive system design. In *IEEE Transactions on Software Engineering*, volume 28, 8, pages 797–813, August 2002.

[40] G. Mori, F. Patern, and C. Santoro. Tool support for designing nomadic applications. In *Proceedings of the 8th international conference on Intelligent user interfaces*, pages 141–148, January 2003.

[41] N.C. Narendra. Adaptive workflow management an integrated approach and system architecture. *SAC00*, pages 858–865, 2000.

[42] N.J. Nunes and P. Campos. Towards usable analysis, design and modeling tools. In *(IUI-CADUI 2004) Workshop: Making Model-based UI Design Practical: Usable and Open Methods and Tool*, January 2004.

[43] J. Pascual, M. Mass, and P.G. Lpez. Model-based design and new user interfaces: Current practices and opportunities. In *(IUI-CADUI 2004) Workshop: Making Model-based UI Design Practical: Usable and Open Methods and Tool*, January 2004.

[44] F. Paterno. *Task Models in Interactive Software Systems*. World Scientific Publishing Company, 2001.

[45] F. Paterno. Tools for task modelling: Where we are, where we are headed. In *Proceedings TAMODIA 2002*, pages 10–17, July 2002.

[46] F. Paterno. Teresa xml, 2004.

[47] F. Paterno and C. Mancini. Developing adaptable hypermedia. In *Proceedings of ACM Intelligent User Interfaces*, pages 163–170. ACM Press, 1999.

[48] F. Paterno, C. Mancini, and S. Meniconi. Concurtasktrees: A diagrammatic notation for specifying task models. In *Proceedings Interact97*, pages 362–369. Chapman and Hall, 1997.

[49] F. Paterno and C. Santoro. One model, many interfaces. In *Proceedings of the 4th International Conference on Design of User Interfaces CADUI2002*, pages 143–154. Kluwer Academic Publishers, May 2002.

[50] D. Petriu and M. Woodside. Analysing software requirments specifications for performance. *WOSP02*, July 2002.

[51] A. Puerta and Eisenstein. XIML: A common representation for interaction data. In *Proceedings of the 7th International Conference on Intelligent User Interfaces*, pages 69–76. ACM Press, January 2002.

[52] A. Puerta and Eisenstein. XIML: A universal language for user interfaces. unpublished work, 2002.

[53] S. Sadiq, M. Orlowska, W. Sadiq, and C. Foulger. Data flow and validation in workflow modelling. In *Proceedings of the fifteenth conference on Australasian database*, pages 207 – 214. Australian Computer Society, Inc., 2004.

[54] K. Siddiqui, D. Petriu, and M. Woodside. Performance-related completions for software specifications. In *Proceedings of the 24th International Conference on Software Engineering*, pages 22–32. ACM Press, May 2002.

[55] I. Singh, B. Stearns, and M. Johnson. *Designing Enterprise Applications with the J2EE Platform*. Addison-Wesley, second edition, 2001.

[56] D. Sinnig, A. Gaffar, A. Seffah, and P. Forbrig. Patterns, tools and models for interaction design. In *(IUI-CADUI 2004) Workshop: Making Model-based UI Design Practical: Usable and Open Methods and Tool*, January 2004.

[57] A. C. Siochi and H.R. Hartson. Task-oriented representation of asynchronous user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Wings for the mind*, pages 183 – 188, 1989.

[58] A. C. Siochi, D. Hix, and H.R. Hartson. *The UAN: A notation to support user-centered design of direct manipulation interfaces*. Academic Press, 1991.

[59] N. Souchon and J. Vanderdonckt. A review of XML-compliant user interface description languages. *DSV-IS2003*, 2003.

[60] N.A. Stavness and K.A. Schneider. Supporting flexible business processes with a progression model. *(IUI-CADUI 2004) Workshop: Making Model-based UI Design Practical: Usable and Open Methods and Tool*, January 2004.

[61] N.A. Stavness and K.A. Schneider. Supporting workflow in user interface description languages. *(AVI 2004)Workshop: Developing User Interfaces with XML: Advances on User Interface Description Languages*, May 2004.

[62] H. Traetteberg. Modeling work: Workflow and task modeling. In *Vanderdonckt, J., Puerta, A.R. (eds.): Proc. of 3 rd Int. Conf. on Computer-Aided Design of User Interfaces CADUI99*, pages 275–280. Kluwer Academic Publishers, October 1999.

[63] Hallvard Trtteberg. Integrating dialog modelling and application development. In *(IUI-CADUI 2004) Workshop: Making Model-based UI Design Practical: Usable and Open Methods and Tool*, January 2004.

[64] WfMC. Workflow management coalition terminology and glossary, WFMC-TC-1011, document status- issue 2.0. In *Specifying Task Models. (Proceedings Interact97*, pages 362–369. Chapman and Hall, June 1997.

[65] WfMC. Workflow process definition interface- xml process definition language. *Doc. No. WFMC-TC-1025*, 2001.

APPENDICES

APPENDIX A. Progression XSD

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!--W3C Schema generated by XMLSpy v2005 sp1 U (http://www.xmlspy.com)-->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
<xs:element name="progression">
  <xs:complexType>
  <xs:sequence>
  <xs:element name="progressionMeta">
    <xs:complexType>
    <xs:sequence>
      <xs:element name="progressionName">
        <xs:simpleType>
        <xs:restriction base="xs:string"/>
        </xs:simpleType>
      </xs:element>
      <xs:element name="progressionTime">
        <xs:simpleType>
        <xs:restriction base="xs:date"/>
        </xs:simpleType>
      </xs:element>
      <xs:element name="progressionID">
         <xs:simpleType>
         <xs:restriction base="xs:string"/>
         </xs:simpleType>
       </xs:element>
     </xs:sequence>
     </xs:complexType>
  </xs:element>
     <xs:element name="transaction">
       <xs:complexType>
       <xs:sequence>
         <xs:element name="transactionMeta">
           <xs:complexType>
           <xs:sequence>
             <xs:element name="transactionID">
               <xs:simpleType>
               <xs:restriction base="xs:string"/>
               </xs:simpleType>
             </xs:element>
           </xs:sequence>
           </xs:complexType>
         </xs:element>
         <xs:element name="transactionSchema">
```

```
        <xs:simpleType>
        <xs:restriction base="xs:string"/>
        </xs:simpleType>
      </xs:element>
  </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="scene" maxOccurs="unbounded">
  <xs:complexType>
  <xs:sequence>
    <xs:element name="sceneMeta">
      <xs:complexType>
      <xs:sequence>
        <xs:element name="sceneName">
          <xs:simpleType>
          <xs:restriction base="xs:string"/>
          </xs:simpleType>
        </xs:element>
        <xs:element name="sceneTime">
          <xs:simpleType>
          <xs:restriction base="xs:string"/>
          </xs:simpleType>
        </xs:element>
        <xs:element name="sceneID">
          <xs:simpleType>
          <xs:restriction base="xs:string"/>
          </xs:simpleType>
        </xs:element>
      </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="userinterface">
      <xs:complexType>
      <xs:sequence>
        <xs:element name="widgetGroup"
        maxOccurs="unbounded">
          <xs:complexType>
          <xs:sequence>
            <xs:element name="widget"
            maxOccurs="unbounded">
              <xs:complexType>
              <xs:choice>
                <xs:element name="JLabel">
                  <xs:simpleType>
                  <xs:restriction base="xs:string"/>
```

```
            </xs:simpleType>
          </xs:element>
          <xs:element name="JTextField">
            <xs:complexType mixed="true">
              <xs:attribute name="isEditable">
                <xs:simpleType>
                <xs:restriction
                base="xs:boolean" />
                </xs:simpleType>
              </xs:attribute>
            </xs:complexType>
          </xs:element>
          <xs:element name="JButton">
            <xs:simpleType>
            <xs:restriction base="xs:string"/>
            </xs:simpleType>
          </xs:element>
        </xs:choice>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    </xs:complexType>
  </xs:element>
 </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="workflow">
 <xs:complexType>
 <xs:sequence>
   <xs:element name="status">
     <xs:complexType>
     <xs:sequence>
       <xs:element name="worker">
         <xs:simpleType>
         <xs:restriction base="xs:string"/>
         </xs:simpleType>
       </xs:element>
       <xs:element name="state">
         <xs:simpleType>
         <xs:restriction base="xs:string">
         <xs:enumeration value=" Inactive "/>
         <xs:enumeration value="Incomplete"/>
         <xs:enumeration value="Complete"/>
         <xs:enumeration value="InProgress"/>
         </xs:restriction>
```

```
     </xs:simpleType>
    </xs:element>
    <xs:element name="next">
      <xs:simpleType>
      <xs:restriction base="xs:string"/>
      </xs:simpleType>
    </xs:element>
    <xs:element name="prev">
      <xs:simpleType>
      <xs:restriction base="xs:string"/>
      </xs:simpleType>
    </xs:element>
  </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="constraints">
  <xs:complexType>
  <xs:sequence>
    <xs:element name="reorder">
      <xs:complexType>
      <xs:sequence>
        <xs:element name="before"
        maxOccurs="unbounded">
          <xs:complexType>
          <xs:sequence>
             <xs:element
             name="errorType">
               <xs:simpleType>
               <xs:restriction
               base="xs:string"/>
               </xs:simpleType>
             </xs:element>
          </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="after"
        maxOccurs="unbounded">
          <xs:complexType>
          <xs:sequence>
            <xs:element
            name="errorType">
              <xs:simpleType>
              <xs:restriction
              base="xs:string"/>
              </xs:simpleType>
```

```
                        </xs:element>
                      </xs:sequence>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      </xs:complexType>
</xs:element>
</xs:schema>
```

APPENDIX B. Sample Transaction XSD used in Travel Expense Approval System Example

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!--W3C Schema generated by XMLSpy v2005 sp1 U (http://www.xmlspy.com)-->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
<xs:element name="transaction">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="contact">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="name">
              <xs:complexType mixed="true">
                <xs:attribute name="status" use="required">
                  <xs:simpleType>
                    <xs:restriction base="xs:string"/>
                  </xs:simpleType>
                </xs:attribute>
              </xs:complexType>
            </xs:element>
            <xs:element name="address">
              <xs:complexType mixed="true">
                <xs:attribute name="status" use="required">
                  <xs:simpleType>
                    <xs:restriction base="xs:string"/>
                  </xs:simpleType>
                </xs:attribute>
              </xs:complexType>
            </xs:element>
            <xs:element name="department">
              <xs:complexType mixed="true">
                <xs:attribute name="status" use="required">
                  <xs:simpleType>
                    <xs:restriction base="xs:string"/>
                  </xs:simpleType>
                </xs:attribute>
              </xs:complexType>
            </xs:element>
            <xs:element name="phone">
              <xs:complexType mixed="true">
                <xs:attribute name="status" use="required">
                  <xs:simpleType>
                    <xs:restriction base="xs:string"/>
```

```
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>
<xs:element name="employeeNum">
  <xs:complexType mixed="true">
    <xs:attribute name="status" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string"/>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>
<xs:element name="purpose">
  <xs:complexType mixed="true">
    <xs:attribute name="status" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string"/>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>
<xs:element name="destination">
  <xs:complexType mixed="true">
    <xs:attribute name="status" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string"/>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>
<xs:element name="conference">
  <xs:complexType mixed="true">
    <xs:attribute name="status" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string"/>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>
<xs:element name="additional">
  <xs:complexType mixed="true">
    <xs:attribute name="status" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string"/>
```

```
            </xs:simpleType>
          </xs:attribute>
        </xs:complexType>
      </xs:element>
      <xs:element name="departDate">
        <xs:complexType mixed="true">
          <xs:attribute name="status" use="required">
            <xs:simpleType>
              <xs:restriction base="xs:string"/>
            </xs:simpleType>
          </xs:attribute>
        </xs:complexType>
      </xs:element>
      <xs:element name="returnDate">
        <xs:complexType mixed="true">
          <xs:attribute name="status" use="required">
            <xs:simpleType>
              <xs:restriction base="xs:string"/>
            </xs:simpleType>
          </xs:attribute>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="expenses">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="vehicle">
        <xs:complexType mixed="true">
          <xs:attribute name="checked" use="required">
            <xs:simpleType>
              <xs:restriction base="xs:boolean"/>
            </xs:simpleType>
          </xs:attribute>
          <xs:attribute name="status" use="required">
            <xs:simpleType>
              <xs:restriction base="xs:string"/>
              </xs:simpleType>
            </xs:attribute>
        </xs:complexType>
      </xs:element>
      <xs:element name="hotel">
        <xs:complexType mixed="true">
          <xs:attribute name="checked" use="required">
```

```
      <xs:simpleType>
        <xs:restriction base="xs:boolean"/>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="status" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string"/>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>
<xs:element name="meals">
  <xs:complexType mixed="true">
    <xs:attribute name="checked" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:boolean"/>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="status" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string"/>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>
<xs:element name="registrationFee">
  <xs:complexType mixed="true">
    <xs:attribute name="checked" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:boolean"/>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="status" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string"/>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>
<xs:element name="entertainment">
  <xs:complexType mixed="true">
    <xs:attribute name="checked" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:boolean"/>
      </xs:simpleType>
```

```
    </xs:attribute>
    <xs:attribute name="status" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string"/>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>
<xs:element name="airFare">
  <xs:complexType mixed="true">
    <xs:attribute name="checked" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:boolean"/>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="status" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string"/>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>
<xs:element name="otherTransport">
  <xs:complexType mixed="true">
    <xs:attribute name="checked" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:boolean"/>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="status" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string"/>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>
<xs:element name="otherExpenses">
  <xs:complexType mixed="true">
    <xs:attribute name="checked" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:boolean"/>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="status" use="required">
      <xs:simpleType>
```

```
            <xs:restriction base="xs:string"/>
          </xs:simpleType>
        </xs:attribute>
      </xs:complexType>
    </xs:element>
    <xs:element name="total">
      <xs:complexType mixed="true">
        <xs:attribute name="checked" use="required">
          <xs:simpleType>
            <xs:restriction base="xs:boolean"/>
          </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="status" use="required">
          <xs:simpleType>
            <xs:restriction base="xs:string"/>
          </xs:simpleType>
        </xs:attribute>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="accounts">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="account">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="code">
       <xs:complexType mixed="true">
        <xs:attribute name="checked" use="required">
         <xs:simpleType>
          <xs:restriction base="xs:boolean"/>
         </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="status" use="required">
         <xs:simpleType>
          <xs:restriction base="xs:string"/>
         </xs:simpleType>
        </xs:attribute>
       </xs:complexType>
      </xs:element>
      <xs:element name="subCode">
       <xs:complexType mixed="true">
        <xs:attribute name="checked" use="required">
```

```
                    <xs:simpleType>
                      <xs:restriction base="xs:boolean"/>
                    </xs:simpleType>
                  </xs:attribute>
                  <xs:attribute name="status" use="required">
                    <xs:simpleType>
                      <xs:restriction base="xs:string"/>
                    </xs:simpleType>
                  </xs:attribute>
                </xs:complexType>
              </xs:element>
              <xs:element name="amount">
                <xs:complexType mixed="true">
                  <xs:attribute name="checked" use="required">
                    <xs:simpleType>
                      <xs:restriction base="xs:boolean"/>
                    </xs:simpleType>
                  </xs:attribute>
                  <xs:attribute name="status" use="required">
                    <xs:simpleType>
                      <xs:restriction base="xs:string"/>
                    </xs:simpleType>
                  </xs:attribute>
                 </xs:complexType>
              </xs:element>
              <xs:element name="gstCode">
                <xs:complexType mixed="true">
                  <xs:attribute name="checked" use="required">
                    <xs:simpleType>
                      <xs:restriction base="xs:boolean"/>
                    </xs:simpleType>
                  </xs:attribute>
                  <xs:attribute name="status" use="required">
                    <xs:simpleType>
                      <xs:restriction base="xs:string"/>
                    </xs:simpleType>
                  </xs:attribute>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

```
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```